## Table of Contents

1. General Project Information ................................................................. 2
2. Why use Apache ActiveMQ Artemis? ...................................................... 3
3. Messaging Concepts .......................................................................... 4
    3.1. General Concepts ................................................................. 4
    3.2. Messaging styles ................................................................. 4
    3.3. Delivery guarantees .............................................................. 6
    3.4. Transactions ......................................................................... 6
    3.5. Durability ................................................................................ 6
    3.6. Messaging APIs ...................................................................... 6
    3.7. High Availability ................................................................. 7
    3.8. Clusters ................................................................................ 7
    3.9. Bridges and routing ............................................................... 8
4. Core Architecture ............................................................................ 9
    4.1. Standalone Broker ................................................................. 10
    4.2. Embedded Broker ................................................................. 10
    4.3. Integrated with a Java/Jakarta EE application server ................. 10
5. Protocols and Interoperability ......................................................... 12
    5.1. Supported Protocols .............................................................. 12
    5.2. Configuring Acceptors ........................................................... 13
6. AMQP ............................................................................................... 15
    6.1. Examples ............................................................................... 15
    6.2. Message Conversions ............................................................ 15
    6.3. Intercepting and changing messages ....................................... 16
    6.4. AMQP and security ............................................................... 16
    6.5. AMQP and destinations ........................................................ 16
    6.6. AMQP and Multicast Addresses (Topics) ............................... 16
    6.7. AMQP and Coordinations - Handling Transactions ............... 17
    6.8. AMQP scheduling message delivery ...................................... 17
    6.9. DLQ and Expiry transfer ......................................................... 17
    6.10. Filtering on Message Annotations ......................................... 18
    6.11. Configuring AMQP Idle Timeout ......................................... 18
    6.12. Web Sockets ....................................................................... 19
7. STOMP ............................................................................................. 20
    7.1. Limitations ............................................................................. 20
    7.2. Mapping STOMP destinations to addresses and queues .......... 20
    7.3. Logging .................................................................................. 20
    7.4. Routing Semantics ............................................................... 21
    7.5. STOMP heart-beating and connection-ttl ............................... 22
12.1. JMS Topic ................................................................. 46
12.2. JMS Queue ............................................................... 46

13. Using JMS or Jakarta Messaging ........................................ 47
13.1. A simple ordering system .............................................. 47
13.2. JNDI ........................................................................ 47
13.3. Directly instantiating JMS Resources without using JNDI .. 52
13.4. Setting The Client ID .................................................. 53
13.5. Setting The Batch Size for DUPS_OK ......................... 54
13.6. Setting The Transaction Batch Size ......................... 54
13.7. Setting The Destination Cache ................................. 54

14. Extra Acknowledge Modes ............................................... 55
14.1. Using PRE_ACKNOWLEDGE .................................... 55
14.2. Individual Acknowledge ............................................ 55
14.3. Example ................................................................. 56

15. Versions ................................................................. 57
15.1. 2.34.0 ..................................................................... 57
15.2. 2.33.0 ..................................................................... 58
15.3. 2.32.0 ..................................................................... 59
15.4. 2.31.2 ..................................................................... 60
15.5. 2.31.1 ..................................................................... 60
15.6. 2.31.0 ..................................................................... 60
15.7. 2.30.0 ..................................................................... 61
15.8. 2.29.0 ..................................................................... 61
15.9. 2.28.0 ..................................................................... 62
15.10. 2.27.1 .................................................................... 63
15.11. 2.27.0 .................................................................... 63
15.12. 2.26.0 .................................................................... 65
15.13. 2.25.0 .................................................................... 65
15.14. 2.24.0 .................................................................... 65
15.15. 2.23.1 .................................................................... 66
15.16. 2.23.0 .................................................................... 66
15.17. 2.22.0 .................................................................... 66
15.18. 2.21.0 .................................................................... 67
15.19. 2.20.0 .................................................................... 68
15.20. 2.19.0 .................................................................... 68
15.21. 2.18.0 .................................................................... 68
15.22. 2.17.0 .................................................................... 70
15.23. 2.16.0 .................................................................... 70
15.24. 2.15.0 .................................................................... 72
15.25. 2.14.0 .................................................................... 72
15.26. 2.13.0 .................................................................... 72
87. Examples

87.1. Running the Examples

87.2. Application-Layer Failover

87.3. Core Bridge Example

87.4. Browser

87.5. Camel

87.6. Client Kickoff

87.7. Client side failover listener

87.8. Client-Side Load-Balancing

87.9. Clustered Durable Subscription

87.10. Clustered Grouping

87.11. Clustered Queue

87.12. Clustering with JGroups

87.13. Clustered Standalone

87.14. Clustered Static Discovery

87.15. Clustered Static Cluster One Way

87.16. Clustered Topic

87.17. Message Consumer Rate Limiting

87.18. Dead Letter

87.19. Delayed Redelivery

87.20. Divert

87.21. Durable Subscription

87.22. Embedded

87.23. Embedded Simple

87.24. Exclusive Queue

87.25. Message Expiration

87.26. Apache ActiveMQ Artemis Resource Adapter example

87.27. HTTP Transport

87.28. Instantiate JMS Objects Directly

87.29. Interceptor

87.30. Interceptor AMQP

87.31. Interceptor Client

87.32. Interceptor MQTT

87.33. JAAS

87.34. JMS Auto Closable

87.35. JMS Completion Listener

87.36. JMS Bridge

87.37. JMS Context

87.38. JMS Shared Consumer

87.39. JMX Management

87.40. Large Message
An in-depth manual on all aspects of Apache ActiveMQ Artemis 2.34.0
Chapter 1. General Project Information

- Apache ActiveMQ Artemis is an open source project to build a multi-protocol, embeddable, very high performance, clustered, asynchronous messaging system.

- Apache ActiveMQ Artemis is an example of Message Oriented Middleware (MoM). For a description of MoMs and other messaging concepts please see the Messaging Concepts.

- If you have any questions related to the use or development of Apache ActiveMQ Artemis please use one of our mailing lists.

- **Official project page:** [https://activemq.apache.org/components/artemis/](https://activemq.apache.org/components/artemis/).

- **Download:** [https://activemq.apache.org/components/artemis/download/](https://activemq.apache.org/components/artemis/download/)

- **Git repository:** [https://github.com/apache/activemq-artemis](https://github.com/apache/activemq-artemis)
Chapter 2. Why use Apache ActiveMQ Artemis?

Here are just a few reasons:

• 100% open source software. Apache ActiveMQ Artemis is licensed using the Apache Software License v 2.0 to minimise barriers to adoption.

• Apache ActiveMQ Artemis is designed with usability in mind.

• Written in Java. Runs on any platform with a Java 11+ runtime, that’s everything from Windows desktops to IBM mainframes.

• Amazing performance. Our ground-breaking high performance journal provides persistent messaging performance at rates normally seen for non-persistent messaging, our non-persistent messaging performance rocks the boat too.

• Full feature set. All the features you’d expect in any serious messaging system, and others you won’t find anywhere else.

• Elegant, clean-cut design with minimal third party dependencies. Run ActiveMQ Artemis stand-alone, run it in integrated in your favourite Java EE application server, or run it embedded inside your own product. It’s up to you.

• Seamless high availability. We provide a HA solution with automatic client failover so you can guarantee zero message loss or duplication in event of server failure.

• Hugely flexible clustering. Create clusters of servers that know how to load balance messages. Link geographically distributed clusters over unreliable connections to form a global network. Configure routing of messages in a highly flexible way.
Apache ActiveMQ Artemis is an asynchronous messaging system, an example of Message Oriented Middleware, we’ll just call them messaging systems in the remainder of this book.

We’ll first present a brief overview of what kind of things messaging systems do, where they’re useful and the kind of concepts you’ll hear about in the messaging world.

If you’re already familiar with what a messaging system is and what it’s capable of, then you can skip this chapter.

3.1. General Concepts

Messaging systems allow you to loosely couple heterogeneous systems together, whilst typically providing reliability, transactions and many other features.

Unlike systems based on a Remote Procedure Call (RPC) pattern, messaging systems primarily use an asynchronous message passing pattern with no tight relationship between requests and responses. Most messaging systems also support a request-response mode but this is not a primary feature of messaging systems.

Designing systems to be asynchronous from end-to-end allows you to really take advantage of your hardware resources, minimizing the amount of threads blocking on IO operations, and to use your network bandwidth to its full capacity. With an RPC approach you have to wait for a response for each request you make so are limited by the network round trip time, or latency of your network. With an asynchronous system you can pipeline flows of messages in different directions, so are limited by the network bandwidth not the latency. This typically allows you to create much higher performance applications.

Messaging systems decouple the senders of messages from the consumers of messages. The senders and consumers of messages are completely independent and know nothing of each other. This allows you to create flexible, loosely coupled systems.

Often, large enterprises use a messaging system to implement a message bus which loosely couples heterogeneous systems together. Message buses often form the core of an Enterprise Service Bus (ESB). Using a message bus to de-couple disparate systems can allow the system to grow and adapt more easily. It also allows more flexibility to add new systems or retire old ones since they don’t have brittle dependencies on each other.

3.2. Messaging styles

Messaging systems normally support two main styles of asynchronous messaging: message queue messaging (also known as point-to-point messaging) and publish subscribe messaging. We’ll summarise them briefly here:

3.2.1. Point-to-Point

With this type of messaging you send a message to a queue. The message is then typically persisted
to provide a guarantee of delivery, then some time later the messaging system delivers the message
to a consumer. The consumer then processes the message and when it is done, it acknowledges the
message. Once the message is acknowledged it disappears from the queue and is not available to be
delivered again. If the system crashes before the messaging server receives an acknowledgement
from the consumer, then on recovery, the message will be available to be delivered to a consumer
again.

With point-to-point messaging, there can be many consumers on the queue but a particular
message will only ever be consumed by a maximum of one of them. Senders (also known as
producers) to the queue are completely decoupled from receivers (also known as consumers) of the
queue - they do not know of each other's existence.

A classic example of point to point messaging would be an order queue in a company's book
ordering system. Each order is represented as a message which is sent to the order queue. Let's
imagine there are many front end ordering systems which send orders to the order queue. When a
message arrives on the queue it is persisted - this ensures that if the server crashes the order is not
lost. Let's also imagine there are many consumers on the order queue - each representing an
instance of an order processing component - these can be on different physical machines but
consuming from the same queue. The messaging system delivers each message to one and only one
of the ordering processing components. Different messages can be processed by different order
processors, but a single order is only processed by one order processor - this ensures orders aren't
processed twice.

As an order processor receives a message, it fulfills the order, sends order information to the
warehouse system and then updates the order database with the order details. Once it's done that it
acknowledges the message to tell the server that the order has been processed and can be forgotten
about. Often the send to the warehouse system, update in database and acknowledgement will be
completed in a single transaction to ensure ACID properties.

3.2.2. Publish-Subscribe

With publish-subscribe messaging many senders can send messages to an entity on the server,
often called a topic (e.g. in the JMS world).

There can be many subscriptions on a topic, a subscription is just another word for a consumer of a
topic. Each subscription receives a copy of each message sent to the topic. This differs from the
message queue pattern where each message is only consumed by a single consumer.

Subscriptions can optionally be durable which means they retain a copy of each message sent to the
topic until the subscriber consumes them - even if the server crashes or is restarted in between.
Non-durable subscriptions only last a maximum of the lifetime of the connection that created them.

An example of publish-subscribe messaging would be a news feed. As news articles are created by
different editors around the world they are sent to a news feed topic. There are many subscribers
around the world who are interested in receiving news items - each one creates a subscription and
the messaging system ensures that a copy of each news message is delivered to each subscription.
3.3. Delivery guarantees

A key feature of most messaging systems is **reliable messaging**. With reliable messaging the server gives a guarantee that the message will be delivered once and only once to each consumer of a queue or each durable subscription of a topic, even in the event of system failure. This is crucial for many businesses; e.g. you don’t want your orders fulfilled more than once or any of your orders to be lost.

In other cases you may not care about a once and only once delivery guarantee and are happy to cope with duplicate deliveries or lost messages - an example of this might be transient stock price updates - which are quickly superseded by the next update on the same stock. The messaging system allows you to configure which delivery guarantees you require.

3.4. Transactions

Messaging systems typically support the sending and acknowledgement of multiple messages in a single local transaction. Apache ActiveMQ Artemis also supports the sending and acknowledgement of message as part of a large global transaction - using the Java mapping of XA: JTA.

3.5. Durability

Messages are either durable or non durable. Durable messages will be persisted in permanent storage and will survive server failure or restart. Non durable messages will not survive server failure or restart. Examples of durable messages might be orders or trades, where they cannot be lost. An example of a non durable message might be a stock price update which is transitory and doesn't need to survive a restart.

3.6. Messaging APIs

How do client applications interact with messaging systems in order to send and consume messages?

Several messaging systems provide their own proprietary APIs with which the client communicates with the messaging system.

There are also some standard ways of operating with messaging systems and some emerging standards in this space. Let’s take a brief look at these.

3.6.1. JMS & Jakarta Messaging

**JMS** was historically part of Oracle’s Java EE specification. However, in 2017 control was transferred to the Eclipse Foundation and it is now known as **Jakarta Messaging** which is part of Jakarta EE.

It is a Java API that encapsulates both message queue and publish-subscribe messaging patterns. It is a lowest common denominator specification - i.e. it was created to encapsulate common functionality of the already existing messaging systems that were available at the time of its creation.
It is a very popular API and is implemented by most messaging systems. It is only available to clients running Java.

It does not define a standard wire format - it only defines a programmatic API so clients and servers from different vendors cannot directly interoperate since each will use the vendor’s own internal wire protocol.

Apache ActiveMQ Artemis provides client implementations which are a fully compliant with JMS 1.1 & 2.0 as well as Jakarta Messaging 2.0 & 3.0.

3.6.2. System specific APIs

Many systems provide their own programmatic API for which to interact with the messaging system. The advantage of this it allows the full set of system functionality to be exposed to the client application. API's like JMS are not normally rich enough to expose all the extra features that most messaging systems provide.

Apache ActiveMQ Artemis provides its own core client API for clients to use if they wish to have access to functionality over and above that accessible via the JMS API.

Please see Core for using the Core API with Apache ActiveMQ Artemis.

3.7. High Availability

High Availability (HA) means that the system should remain operational after failure of one or more of the servers. The degree of support for HA varies between various messaging systems.

Apache ActiveMQ Artemis provides automatic failover where your sessions are automatically reconnected to a backup server on event of a server failure.

For more information on HA, please see High Availability and Failover.

3.8. Clusters

Many messaging systems allow you to create groups of messaging servers called clusters. Clusters allow the load of sending and consuming messages to be spread over many servers. This allows your system to scale horizontally by adding new servers to the cluster.

Degrees of support for clusters varies between messaging systems, with some systems having fairly basic clusters with the cluster members being hardly aware of each other.

Apache ActiveMQ Artemis provides very configurable state-of-the-art clustering model where messages can be intelligently load balanced between the servers in the cluster, according to the number of consumers on each node, and whether they are ready for messages.

Apache ActiveMQ Artemis also has the ability to automatically redistribute messages between nodes of a cluster to prevent starvation on any particular node.

For full details on clustering, please see Clusters.
3.9. Bridges and routing

Some messaging systems allow isolated clusters or single nodes to be bridged together, typically over unreliable connections like a wide area network (WAN), or the internet.

A bridge normally consumes from a queue on one server and forwards messages to another queue on a different server. Bridges cope with unreliable connections, automatically reconnecting when the connections becomes available again.

Apache ActiveMQ Artemis bridges can be configured with filter expressions to only forward certain messages, and transformation can also be hooked in.

Apache ActiveMQ Artemis also allows routing between queues to be configured in server side configuration. This allows complex routing networks to be set up forwarding or copying messages from one destination to another, forming a global network of interconnected brokers.

For more information please see Core Bridges and Diverting and Splitting Message Flows.
Chapter 4. Core Architecture

Apache ActiveMQ Artemis core is designed simply as a set of Plain Old Java Objects (POJOs) - we hope you like its clean-cut design.

Each Apache ActiveMQ Artemis server has its own ultra high performance persistent journal, which it uses for message and other persistence.

Using a high performance journal allows outrageous persistence message performance, something not achievable when using a relational database for persistence (although JDBC is still an option if necessary).

Apache ActiveMQ Artemis clients, potentially on different physical machines, interact with the Apache ActiveMQ Artemis broker. Apache ActiveMQ Artemis currently ships three API implementations for messaging at the client side:

1. Core client API. This is a simple intuitive Java API that is aligned with the Artemis internal Core. Allowing more control of broker objects (e.g. direct creation of addresses and queues). The Core API also offers a full set of messaging functionality without some of the complexities of JMS.

2. JMS 2.0 client API. The standard JMS API is available at the client side. This client is also compliant with the Jakarta Messaging 2.0 specification.

3. Jakarta Messaging 3.0 client API. This is essentially the same as the JMS 2.0 API. The only difference is the package names use jakarta instead of javax. This difference was introduced due to the move from Oracle's Java EE to Eclipse's Jakarta EE.

Apache ActiveMQ Artemis also provides different protocol implementations on the server so you can use respective clients for these protocols:

- AMQP
- OpenWire
- MQTT
- STOMP
- HornetQ (for use with HornetQ clients).
- Core (Artemis CORE protocol)

JMS semantics are implemented by a JMS facade layer on the client side.

The Apache ActiveMQ Artemis broker does not speak JMS and in fact does not know anything about JMS, it is a protocol agnostic messaging server designed to be used with multiple different protocols.

When a user uses the JMS API on the client side, all JMS interactions are translated into operations on the Apache ActiveMQ Artemis core client API before being transferred over the wire using the core protocol.

The broker always just deals with core API interactions.
4.1. Standalone Broker

The normal stand-alone messaging broker configuration comprises a core messaging broker and a number of protocol managers that provide support for the various protocol mentioned earlier.

The standalone broker configuration uses picocli for bootstrapping the broker.

For more information on server configuration files see Server Configuration.

4.2. Embedded Broker

Apache ActiveMQ Artemis core is designed as a set of simple POJOs so if you have an application that requires messaging functionality internally but you don't want to expose that as an Apache ActiveMQ Artemis broker you can directly instantiate and embed brokers in your own application.

Read more about embedding Apache ActiveMQ Artemis.

4.3. Integrated with a Java/Jakarta EE application server

Apache ActiveMQ Artemis provides its own fully functional Java Connector Architecture (JCA) adaptor which enables it to be integrated easily into any Java/Jakarta EE (henceforth just "EE") compliant application server or servlet engine.

EE application servers provide Message Driven Beans (MDBs), which are a special type of Enterprise Java Beans (EJBs) that can process messages from sources such as JMS systems or mail systems.

Probably the most common use of an MDB is to consume messages from a JMS messaging system.

According to the EE specification an application server uses a JCA adapter to integrate with a JMS messaging system so it can consume messages for MDBs.

However, the JCA adapter is not only used by the EE application server for consuming messages via MDBs, it is also used when sending message to the JMS messaging system e.g. from inside an EJB or servlet.

When integrating with a JMS messaging system from inside an EE application server it is always recommended that this is done via a JCA adaptor. In fact, communicating with a JMS messaging system directly, without using JCA would be illegal according to the EE specification.

The application server’s JCA service provides extra functionality such as connection pooling and automatic transaction enlistment, which are desirable when using messaging, say, from inside an EJB. It is possible to talk to a JMS messaging system directly from an EJB, MDB or servlet without going through a JCA adapter, but this is not recommended since you will not be able to take advantage of the JCA features, such as caching of JMS sessions, which can result in poor performance.

Note that all communication between EJB sessions or entity beans and Message Driven beans go
through the adaptor and not directly to Apache ActiveMQ Artemis.

The large arrow with the prohibited sign shows an EJB session bean talking directly to the Apache ActiveMQ Artemis server. This is not recommended as you'll most likely end up creating a new connection and session every time you want to interact from the EJB, which is an anti-pattern.
Chapter 5. Protocols and Interoperability

Apache ActiveMQ Artemis has a powerful & flexible core which provides a foundation upon which other protocols can be implemented. Each protocol implementation translates the ideas of its specific protocol onto this core.

The broker ships with a client implementation which interacts directly with this core. It uses what’s called the “core” API, and it communicates over the network using the "core" protocol.

5.1. Supported Protocols

The broker has a pluggable protocol architecture. Protocol plugins come in the form of protocol modules. Each protocol module is included on the broker's class path and loaded by the broker at boot time. The broker ships with 5 protocol modules out of the box.

5.1.1. AMQP

AMQP is a specification for interoperable messaging. It also defines a wire format, so any AMQP client can work with any messaging system that supports AMQP. AMQP clients are available in many different programming languages.

Apache ActiveMQ Artemis implements the AMQP 1.0 specification. Any client that supports the 1.0 specification will be able to interact with Apache ActiveMQ Artemis.

Please see AMQP for more details.

5.1.2. MQTT

MQTT is a lightweight connectivity protocol. It is designed to run in environments where device and networks are constrained. Any client that supports the 3.1, 3.1.1, or 5 specification will be able to interact with Apache ActiveMQ Artemis.

Please see MQTT for more details.

5.1.3. STOMP

Stomp is a very simple text protocol for interoperating with messaging systems. It defines a wire format, so theoretically any Stomp client can work with any messaging system that supports Stomp. Stomp clients are available in many different programming languages. Any client that supports the 1.0, 1.1, or 1.2 specification will be able to interact with Apache ActiveMQ Artemis.

Please see Stomp for more details.

5.1.4. OpenWire

ActiveMQ Classic defines its own wire protocol: OpenWire. In order to support ActiveMQ "Classic" clients, Apache ActiveMQ Artemis supports OpenWire. Any ActiveMQ 5.12.x or higher can be used with Apache ActiveMQ Artemis.
Please see OpenWire for more details.

5.1.5. Core

ActiveMQ Classic defines its own wire protocol: OpenWire. In order to support ActiveMQ "Classic" clients, Apache ActiveMQ Artemis supports OpenWire. Any ActiveMQ 5.12.x or higher can be used with Apache ActiveMQ Artemis.

Please see Core for more details.

APIs and Other Interfaces

Although JMS and Jakarta Messaging are standardized APIs, they do not define a network protocol. The ActiveMQ Artemis JMS & Jakarta Messaging clients are implemented on top of the core protocol. We also provide a client-side JNDI implementation.

5.2. Configuring Acceptors

In order to make use of a particular protocol, a transport must be configured with the desired protocol enabled. There is a whole section on configuring transports that can be found here.

The default configuration shipped with the ActiveMQ Artemis distribution comes with a number of acceptors already defined, one for each of the above protocols plus a generic acceptor that supports all protocols. To enable protocols on a particular acceptor simply add the protocols url parameter to the acceptor url where the value is one or more protocols (separated by commas). If the protocols parameter is omitted from the url all protocols are enabled.

• The following example enables only MQTT on port 1883

```
<acceptors>
  <acceptor>tcp://localhost:1883?protocols=MQTT</acceptor>
</acceptors>
```

• The following example enables MQTT and AMQP on port 5672

```
<acceptors>
  <acceptor>tcp://localhost:5672?protocols=MQTT,AMQP</acceptor>
</acceptors>
```

• The following example enables all protocols on 61616:

```
<acceptors>
  <acceptor>tcp://localhost:61616</acceptor>
</acceptors>
```

Here are the supported protocols and their corresponding value used in the protocols url
parameter.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>protocols value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core (Artemis &amp; HornetQ native)</td>
<td>CORE</td>
</tr>
<tr>
<td>OpenWire (5.x native)</td>
<td>OPENWIRE</td>
</tr>
<tr>
<td>AMQP</td>
<td>AMQP</td>
</tr>
<tr>
<td>MQTT</td>
<td>MQTT</td>
</tr>
<tr>
<td>STOMP</td>
<td>STOMP</td>
</tr>
</tbody>
</table>
Chapter 6. AMQP

Apache ActiveMQ Artemis supports the AMQP 1.0 specification. By default there are acceptor elements configured to accept AMQP connections on ports 61616 and 5672.

See the general Protocols and Interoperability chapter for details on configuring an acceptor for AMQP.

You can use any AMQP 1.0 compatible clients.

A short list includes:

- qpid clients
- .NET Clients
- Javascript NodeJS
- Java Script RHEA
- ... and many others.

6.1. Examples

We have a few examples as part of the Artemis examples:

- .NET:
  -./examples/protocols/amqp/dotnet
- ProtonCPP
  -./examples/protocols/amqp/proton-cpp
  -./examples/protocols/amqp/proton-clustered-cpp
- Ruby
  -./examples/protocols/amqp/proton-ruby
- Java (Using the qpid JMS Client)
  -./examples/protocols/amqp/queue
- Interceptors
  -./examples/features/standard/interceptor-amqp
  -./examples/features/standard/broker-plugin

6.2. Message Conversions

The broker will not perform any message conversion to any other protocols when sending AMQP and receiving AMQP.

However if you intend your message to be received by an AMQP JMS Client, you must follow the JMS Mapping Conventions. If you send a body type that is not recognized by this specification the
conversion between AMQP and any other protocol will make it a Binary Message. Make sure you follow these conventions if you intend to cross protocols or languages. Especially on the message body.

A compatibility setting allows aligning the naming convention of AMQP queues (JMS Durable and Shared Subscriptions) with CORE. For backwards compatibility reasons, you need to explicitly enable this via broker configuration:

```
amqp-use-core-subscription-naming
• true - use queue naming convention that is aligned with CORE.
• false (default) - use older naming convention.
```

### 6.3. Intercepting and changing messages

We don’t recommend changing messages at the server’s side for a few reasons:

- AMQP messages are meant to be immutable
- The message won’t be the original message the user sent
- AMQP has the possibility of signing messages. The signature would be broken.
- For performance reasons. We try not to re-encode (or even decode) messages.

If regardless these recommendations you still need and want to intercept and change AMQP messages, look at the aforementioned interceptor examples.

### 6.4. AMQP and security

The Apache ActiveMQ Artemis Server accepts the PLAIN, ANONYMOUS, and GSSAPI SASL mechanism. These are implemented on the broker’s security infrastructure.

### 6.5. AMQP and destinations

If an AMQP Link is dynamic then a temporary queue will be created and either the remote source or remote target address will be set to the name of the temporary queue. If the Link is not dynamic then the address of the remote target or source will be used for the queue. In case it does not exist, it will be auto-created if the settings allow.

### 6.6. AMQP and Multicast Addresses (Topics)

Although AMQP has no notion of “topics” it is still possible to treat AMQP consumers or receivers as subscriptions rather than just consumers on a queue. By default any receiving link that attaches to an address that has only multicast enabled will be treated as a subscription and a corresponding subscription queue will be created. If the Terminus Durability is either UNSETTLED_STATE or CONFIGURATION then the queue will be made durable (similar to a JMS durable subscription) and given a name made up from the container id and the link name, something like my-container-id:my-link-name. If the Terminus Durability is configured as NONE then a volatile multicast queue will be created.
6.7. AMQP and Coordinations - Handling Transactions

An AMQP links target can also be a Coordinator. A Coordinator is used to handle transactions. If a coordinator is used then the underlying server session will be transacted and will be either rolled back or committed via the coordinator.

AMQP allows the use of multiple transactions per session, `amqp:multi-txns-per-ssn`, however in this version of Apache ActiveMQ Artemis will only support single transactions per session.

6.8. AMQP scheduling message delivery

An AMQP message can provide scheduling information that controls the time in the future when the message will be delivered at the earliest. This information is provided by adding a message annotation to the sent message.

There are two different message annotations that can be used to schedule a message for later delivery:

**x-opt-delivery-time**

The specified value must be a positive long corresponding to the time the message should be made available for delivery (in milliseconds).

**x-opt-delivery-delay**

The specified value must be a positive long corresponding to the amount of milliseconds after the broker receives the given message before it should be made available for delivery.

If both annotations are present in the same message then the broker will prefer the more specific `x-opt-delivery-time` value.

6.9. DLQ and Expiry transfer

AMQP Messages will be copied before transferred to a DLQ or ExpiryQueue and will receive properties and annotations during this process.

The broker also keeps an internal only property (called extra property) that is not exposed to the clients, and those will also be filled during this process.

Here is a list of Annotations and Property names AMQP Messages will receive when transferred:

<table>
<thead>
<tr>
<th>Annotation name</th>
<th>Internal Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-opt-ORIG-MESSAGE-ID</td>
<td>_AMQ_ORIG_MESSAGE_ID</td>
<td>The original message ID before the transfer</td>
</tr>
<tr>
<td>x-opt-ACTUAL-EXPIRY</td>
<td>_AMQ_ACTUAL_EXPIRY</td>
<td>When the expiry took place. Milliseconds since epoch times</td>
</tr>
</tbody>
</table>
### 6.10. Filtering on Message Annotations

It is possible to filter on messaging annotations if you use the prefix "m." before the annotation name.

For example if you want to filter messages sent to a specific destination, you could create your filter accordingly to this:

```java
ConnectionFactory factory = new JmsConnectionFactory("amqp://localhost:5672");
Connection connection = factory.createConnection();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
connection.start();
javax.jms.Queue queue = session.createQueue("my-DLQ");
MessageConsumer consumer = session.createConsumer(queue, "\"m.x-opt-ORIG-ADDRESS\"='ORIGINAL_PLACE'" );
Message message = consumer.receive();
```

The broker will set internal properties. If you intend to filter after DLQ or Expiry you may choose the internal property names:

```java
// Replace the consumer creation on the previous example:
MessageConsumer consumer = session.createConsumer(queue, 
"_AMQ.ORIG_ADDRESS='ORIGINAL_PLACE'" );
```

### 6.11. Configuring AMQP Idle Timeout

It is possible to configure the AMQP Server’s IDLE Timeout by setting the property amqpIdleTimeout in milliseconds on the acceptor.

This will make the server to send an AMQP frame open to the client, with your configured timeout / 2.

So, if you configured your AMQP Idle Timeout to be 60000, the server will tell the client to send frames every 30,000 milliseconds.

```xml
<acceptor name="amqp">.... ;amqpIdleTimeout=<configured-timeout>; ..... </acceptor>
```
6.11.1. Disabling Keep alive checks

if you set amqpIdleTimeout=0 that will tell clients to not sending keep alive packets towards the server. On this case you will rely on TCP to determine when the socket needs to be closed.

```xml
<acceptor name="amqp">.... ;amqpIdleTimeout=0; ..... </acceptor>
```

This contains a real example for configuring amqpIdleTimeout:

```xml
<acceptor name="amqp">
tcp://0.0.0.0:5672?amqpIdleTimeout=0;tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=AMQP;useEpoll=true;amqpCredits=1000;amqpLowCredits=300;directDeliver=false;batchDelay=10</acceptor>
```

6.12. Web Sockets

Apache ActiveMQ Artemis also supports AMQP over Web Sockets. Modern web browsers which support Web Sockets can send and receive AMQP messages.

AMQP over Web Sockets is supported via a normal AMQP acceptor:

```xml
<acceptor name="amqp-ws-acceptor">
tcp://localhost:5672?protocols=AMQP</acceptor>
```

With this configuration, Apache ActiveMQ Artemis will accept AMQP connections over Web Sockets on the port 5672. Web browsers can then connect to `ws://<server>:5672` using a Web Socket to send and receive AMQP messages.
Chapter 7. STOMP

STOMP is a text-orientated wire protocol that allows STOMP clients to communicate with STOMP Brokers. Apache ActiveMQ Artemis supports STOMP 1.0, 1.1 and 1.2.

STOMP clients are available for several languages and platforms making it a good choice for interoperability.

By default there are acceptor elements configured to accept STOMP connections on ports 61616 and 61613.

See the general Protocols and Interoperability chapter for details on configuring an acceptor for STOMP.

Refer to the STOMP examples for a look at some of this functionality in action.

7.1. Limitations

7.1.1. Transactional Acknowledgements

The STOMP specification identifies transactional acknowledgements as an optional feature. Support for transactional acknowledgements is not implemented in Apache ActiveMQ Artemis. The ACK frame can not be part of a transaction. It will be ignored if its transaction header is set.

7.1.2. Virtual Hosting

Apache ActiveMQ Artemis currently doesn’t support virtual hosting, which means the host header in CONNECT frame will be ignored.

7.2. Mapping STOMP destinations to addresses and queues

STOMP clients deals with destinations when sending messages and subscribing. Destination names are simply strings which are mapped to some form of destination on the server - how the server translates these is left to the server implementation.

In Apache ActiveMQ Artemis, these destinations are mapped to addresses and queues depending on the operation being done and the desired semantics (e.g. anycast or multicast).

7.3. Logging

Incoming and outgoing STOMP frames can be logged by enabling DEBUG for org.apache.activemq.artemis.core.protocol.stomp.StompConnection. This can be extremely useful for debugging or simply monitoring client activity. Along with the STOMP frame itself the remote IP address of the client is logged as well as the internal connection ID so that frames from the same client can be correlated.
7.4. Routing Semantics

The STOMP specification is intentionally ambiguous about message routing semantics. When providing an overview of the protocol the STOMP 1.2 specification says:

A STOMP server is modelled as a set of destinations to which messages can be sent. The STOMP protocol treats destinations as opaque string and their syntax is server implementation specific. Additionally STOMP does not define what the delivery semantics of destinations should be. The delivery, or "message exchange", semantics of destinations can vary from server to server and even from destination to destination. This allows servers to be creative with the semantics that they can support with STOMP.

Therefore, there are a handful of different ways to specify which semantics are desired both on the client-side and broker-side.

7.4.1. Configuring Routing Semantics from the Client Side

Sending

When a STOMP client sends a message (using a SEND frame), the protocol manager looks at the destination-type header to determine where to route it and potentially how to create the address and/or queue to which it is being sent. Valid values are ANYCAST and MULTICAST (case sensitive). If no indication of routing type is supplied (either by the client or the broker) then the default defined in the corresponding default-address-routing-type & default-queue-routing-type address-settings will be used as necessary.

The destination header maps to an address of the same name if MULTICAST is used and additionally to a queue of the same name if ANYCAST is used.

Subscribing

When a STOMP client subscribes to a destination (using a SUBSCRIBE frame), the protocol manager looks at the subscription-type header frame to determine what subscription semantics to use and potentially how to create the address and/or queue for the subscription. If no indication of routing type is supplied (either by the client or the broker) then the default defined in the corresponding default-address-routing-type & default-queue-routing-type address-settings will be used as necessary.

The destination header maps to an address of the same name if MULTICAST is used and additionally to a queue of the same name if ANYCAST is used.

7.4.2. Configuring Routing Semantics from the Broker side

On the broker-side there are two main options for specifying routing semantics - prefixes and
Prefixes

Using prefixes involves specifying the `anycastPrefix` and/or the `multicastPrefix` on the acceptor which the STOMP client is using. For the STOMP use-case these prefixes tell the broker that destinations using them should be treated as anycast or multicast. For example, if the acceptor has `anycastPrefix=queue/` then when a STOMP client sends a message to `destination:queue/foo` the broker will auto-create the address `foo` and queue `foo` appropriately as anycast and the message will be placed in that queue. Additionally, if the acceptor has `multicastPrefix=topic/` then when a STOMP client sends a message to `destination:topic/bar` the broker will auto-create the address `bar` as multicast, but it won't create a queue since multicast (i.e. pub/sub) semantics require a client to explicitly create a subscription to receive those messages.

The `anycastPrefix` and/or `multicastPrefix` on the acceptor will be stripped from the destination value.

Address Settings

Using address settings involves defining address-setting elements whose `match` corresponds with the destination names the clients will use along with the proper `delimiter` to enabled matching. For example, broker.xml could use the following:

```
<address-settings>
  <address-setting match="queue/#">
    <default-address-routing-type>ANYCAST</default-address-routing-type>
    <default-queue-routing-type>ANYCAST</default-queue-routing-type>
  </address>
  <address-setting match="topic/#">
    <default-address-routing-type>MULTICAST</default-address-routing-type>
    <default-queue-routing-type>MULTICAST</default-queue-routing-type>
  </address>
</address-settings>
```

Then if a STOMP client sends a message to `destination:queue/foo` the broker will auto-create the address `queue/foo` and queue `queue/foo` appropriately as anycast and the message will be placed in that queue. Additionally, if a STOMP client sends a message to `destination:topic/bar` the broker will auto-create the address `topic/bar` as multicast, but it won't create a queue as previously explained.

7.5. STOMP heart-beating and connection-ttl

Well behaved STOMP clients will always send a `DISCONNECT` frame before closing their connections. In this case the server will clear up any server side resources such as sessions and consumers synchronously. However if STOMP clients exit without sending a `DISCONNECT` frame or if they crash the server will have no way of knowing immediately whether the client is still alive or not. STOMP
connections therefore default to a connection-ttl value of 1 minute (see chapter on connection-ttl for more information. This value can be overridden using the connection-ttl-override property or if you need a specific connectionTtl for your stomp connections without affecting the broker-wide connection-ttl-override setting, you can configure your stomp acceptor with the connectionTtl property, which is used to set the ttl for connections that are created from that acceptor. For example:

```xml
<acceptor name="stomp-acceptor"
tcp://localhost:61613?protocols=STOMP;connectionTtl=20000</acceptor>
```

The above configuration will make sure that any STOMP connection that is created from that acceptor and does not include a heart-beat header or disables client-to-server heart-beats by specifying a 0 value will have its connection-ttl set to 20 seconds. The connectionTtl set on an acceptor will take precedence over connection-ttl-override. The default connectionTtl is 60,000 milliseconds.

Since STOMP 1.0 does not support heart-beating then all connections from STOMP 1.0 clients will have a connection TTL imposed upon them by the broker based on the aforementioned configuration options. Likewise, any STOMP 1.1 or 1.2 clients that don't specify a heart-beat header or disable client-to-server heart-beating (e.g. by sending 0,X in the heart-beat header) will have a connection TTL imposed upon them by the broker.

For STOMP 1.1 and 1.2 clients which send a non-zero client-to-server heart-beat header value then their connection TTL will be set accordingly. However, the broker will not strictly set the connection TTL to the same value as the specified in the heart-beat since even small network delays could then cause spurious disconnects. Instead, the client-to-server value in the heart-beat will be multiplied by the heartbeatToConnectionTtlModifier specified on the acceptor. The heartbeatToConnectionTtlModifier is a decimal value that defaults to 2.0 so for example, if a client sends a heart-beat header of 1000,0 the connection TTL will be set to 2000 so that the data or ping frames sent every 1000 milliseconds will have a sufficient cushion so as not to be considered late and trigger a disconnect. This is also in accordance with the STOMP 1.1 and 1.2 specifications which both state, "because of timing inaccuracies, the receiver SHOULD be tolerant and take into account an error margin."

The minimum and maximum connection TTL allowed can also be specified on the acceptor via the connectionTtlMin and connectionTtlMax properties respectively. The default connectionTtlMin is 1000 and the default connectionTtlMax is Java’s Long.MAX_VALUE meaning there essentially is no max connection TTL by default. Keep in mind that the heartbeatToConnectionTtlModifier is relevant here. For example, if a client sends a heart-beat header of 20000,0 and the acceptor is using a connectionTtlMax of 30000 and a default heartbeatToConnectionTtlModifier of 2.0 then the connection TTL would be 40000 (i.e. 20000 * 2.0) which would exceed the connectionTtlMax. In this case the server would respond to the client with a heart-beat header of 0,15000 (i.e. 30000 / 2.0). As described previously, this is to make sure there is a sufficient cushion for the client heart-beats in accordance with the STOMP 1.1 and 1.2 specifications. The same kind of calculation is done for connectionTtlMin.

The minimum server-to-client heart-beat value is 500ms.
Please note that the STOMP protocol version 1.0 does not contain any heart-beat frame. It is therefore the user’s responsibility to make sure data is sent within connection-ttl or the server will assume the client is dead and clean up server side resources. With STOMP 1.1 users can use heart-beats to maintain the life cycle of stomp connections.

### 7.6. Selector/Filter expressions

STOMP subscribers can specify an expression used to select or filter what the subscriber receives using the `selector` header. The filter expression syntax follows the core filter syntax described in the Filter Expressions documentation.

### 7.7. STOMP and JMS interoperability

#### 7.7.1. Sending and consuming STOMP message from JMS or Core API

STOMP is mainly a text-orientated protocol. To make it simpler to interoperate with JMS and Core API, our STOMP implementation checks for presence of the `content-length` header to decide how to map a STOMP 1.0 message to a JMS Message or a Core message.

If the STOMP 1.0 message does not have a `content-length` header, it will be mapped to a JMS `TextMessage` or a Core message with a single nullable `SimpleString in the body buffer`.

Alternatively, if the STOMP 1.0 message has a `content-length` header, it will be mapped to a JMS `BytesMessage` or a Core message with a `byte[] in the body buffer`.

The same logic applies when mapping a JMS message or a Core message to STOMP. A STOMP 1.0 client can check the presence of the `content-length` header to determine the type of the message body (String or bytes).

#### 7.7.2. Message IDs for STOMP messages

When receiving STOMP messages via a JMS consumer or a QueueBrowser, the messages have no properties like JMSMessageID by default. However this may bring some inconvenience to clients who wants an ID for their purpose. The broker STOMP provides a parameter to enable message ID on each incoming STOMP message. If you want each STOMP message to have a unique ID, just set the `stompEnableMessageId` to true. For example:

```xml
<acceptor name="stomp-acceptor"
tcp://localhost:61613?protocols=STOMP;stompEnableMessageId=true</acceptor>
```

When the server starts with the above setting, each stomp message sent through this acceptor will have an extra property added. The property key is `amqMessageId` and the value is a String representation of a long type internal message id prefixed with STOMP, like:
The default `stompEnableMessageId` value is false.

## 7.8. Durable Subscriptions

The `SUBSCRIBE` and `UNSUBSCRIBE` frames can be augmented with special headers to create and destroy durable subscriptions respectively.

To create a durable subscription the `client-id` header must be set on the `CONNECT` frame and the `durable-subscription-name` must be set on the `SUBSCRIBE` frame. The combination of these two headers will form the identity of the durable subscription.

To delete a durable subscription the `client-id` header must be set on the `CONNECT` frame and the `durable-subscription-name` must be set on the `UNSUBSCRIBE` frame. The values for these headers should match what was set on the `SUBSCRIBE` frame to delete the corresponding durable subscription.

Aside from `durable-subscription-name`, the broker also supports `durable-subscriber-name` (a deprecated property used before `durable-subscription-name`) as well as `activemq.subscriptionName` from ActiveMQ Classic. This is the order of precedence if the frame contains more than one of these:

1. `durable-subscription-name`
2. `durable-subscriber-name`
3. `activemq.subscriptionName`

It is possible to pre-configure durable subscriptions since the STOMP implementation creates the queue used for the durable subscription in a deterministic way (i.e. using the format of `client-id.subscription-name`). For example, if you wanted to configure a durable subscription on the address `myAddress` with a client-id of `myclientid` and a subscription name of `mysubscription` then configure the durable subscription:

```xml
<addresses>
  <address name="myAddress">
    <multicast>
      <queue name="myclientid.mysubscription"/>
    </multicast>
  </address>
</addresses>
```

## 7.9. Handling of Large Messages with STOMP

STOMP clients may send very large frame bodies which can exceed the size of the broker's internal buffer, causing unexpected errors. To prevent this situation from happening, the broker provides a STOMP configuration attribute `stompMinLargeMessageSize`. This attribute can be configured inside a
stomp acceptor, as a parameter. For example:

```xml
<acceptor name="stomp-acceptor">
  tcp://localhost:61613?protocols=STOMP;stompMinLargeMessageSize=10240
</acceptor>
```

The type of this attribute is integer. When this attributed is configured, the broker will check the size of the body of each STOMP frame arrived from connections established with this acceptor. If the size of the body is equal or greater than the value of `stompMinLargeMessageSize`, the message will be persisted as a large message. When a large message is delivered to a STOMP consumer, the broker will automatically handle the conversion from a large message to a normal message, before sending it to the client.

If a large message is compressed, the server will uncompressed it before sending it to stomp clients. The default value of `stompMinLargeMessageSize` is the same as the default value of `minLargeMessageSize`.

### 7.10. Web Sockets

Apache ActiveMQ Artemis also supports STOMP over Web Sockets. Modern web browsers which support Web Sockets can send and receive STOMP messages.

STOMP over Web Sockets is supported via the normal STOMP acceptor:

```xml
<acceptor name="stomp-ws-acceptor">
  tcp://localhost:61614?protocols=STOMP
</acceptor>
```

With this configuration, Apache ActiveMQ Artemis will accept STOMP connections over Web Sockets on the port 61614. Web browsers can then connect to `ws://<server>:61614` using a Web Socket to send and receive STOMP messages.

A companion JavaScript library to ease client-side development is available from GitHub (please see its documentation for a complete description).

The payload length of Web Socket frames can vary between client implementations. By default the broker will accept frames with a payload length of 65,536. If the client needs to send payloads longer than this in a single frame this length can be adjusted by using the `webSocketMaxFramePayloadLength` URL parameter on the acceptor. In previous version this was configured via the similarly named `stompMaxFramePayloadLength` acceptor URL parameter.

Web Socket frames can be encoded as either binary or text. By default the broker encodes them as binary. However, this can be changed by using the `webSocketEncoderType` acceptor URL parameter. Valid values are binary and text.

The [stomp-websockets example](#) shows how to configure an Apache ActiveMQ Artemis broker to have web browsers and Java applications exchanges messages.
7.11. Flow Control

STOMP clients can use the `consumer-window-size` header on the `SUBSCRIBE` frame to control the flow of messages to clients. This is broadly discussed in the Flow Control chapter.

This ability is similar to the `activemq.prefetchSize` header supported by ActiveMQ "Classic". However, that header specifies the size in terms of messages whereas `consumer-window-size` specifies the size in terms of bytes. ActiveMQ Artemis supports the `activemq.prefetchSize` header for backwards compatibility but the value will be interpreted as bytes just like `consumer-window-size` would be. If both `activemq.prefetchSize` and `consumer-window-size` are set then the value for `consumer-window-size` will be used.

Setting `consumer-window-size` to 0 will ensure that once a STOMP client receives a message that it will not receive another one until it sends the appropriate ACK or NACK frame for the message it already has.

Setting `consumer-window-size` to a value greater than 0 will allow it to receive messages until the cumulative bytes of those messages reaches the configured size. Once that happens the client will not receive any more messages until it sends the appropriate ACK or NACK frame for the messages it already has.

Setting `consumer-window-size` to -1 means there is no flow control and the broker will dispatch messages to clients as fast as it can.

Flow control can be configured at the acceptor as well using the `stompConsumerWindowSize` URL parameter. This value is 10240 (i.e. 10K) by default for clients using client and client-individual acknowledgement modes. It is -1 for clients using the auto acknowledgement mode. Even if `stompConsumerWindowSize` is set on the STOMP acceptor it will be overridden by the value provided by individual clients using the `consumer-window-size` header on their `SUBSCRIBE` frame.

> The `stompConsumerWindowSize` URL parameter used to be called `stompConsumerCredits` but was changed to be more consistent with the new header name (i.e. `consumer-window-size`). The `stompConsumerCredits` parameter is deprecated but it will still work for the time being.

Using the DEBUG logging mentioned earlier it is possible to see the size of the MESSAGE frames dispatched to clients. This can help when trying to determine the best `consumer-window-size` setting.
Chapter 8. MQTT

MQTT is a light weight, client to server, publish / subscribe messaging protocol. MQTT has been specifically designed to reduce transport overhead (and thus network traffic) and code footprint on client devices. For this reason MQTT is ideally suited to constrained devices such as sensors and actuators and is quickly becoming the defacto standard communication protocol for IoT.

Apache ActiveMQ Artemis supports the following MQTT versions (with links to their respective specifications):

- 3.1
- 3.1.1
- 5.0

By default there are acceptor elements configured to accept MQTT connections on ports 61616 and 1883.

See the general Protocols and Interoperability chapter for details on configuring an acceptor for MQTT.

Refer to the MQTT examples for a look at some of this functionality in action.

8.1. MQTT Quality of Service

MQTT offers 3 quality of service levels.

Each message (or topic subscription) can define a quality of service that is associated with it. The quality of service level defined on a topic is the maximum level a client is willing to accept. The quality of service level on a message is the desired quality of service level for this message. The broker will attempt to deliver messages to subscribers at the highest quality of service level based on what is defined on the message and topic subscription.

Each quality of service level offers a level of guarantee by which a message is sent or received:

- **QoS 0: AT MOST ONCE**
  
  Guarantees that a particular message is only ever received by the subscriber a maximum of one time. This does mean that the message may never arrive. The sender and the receiver will attempt to deliver the message, but if something fails and the message does not reach its destination (say due to a network connection) the message may be lost. This QoS has the least network traffic overhead and the least burden on the client and the broker and is often useful for telemetry data where it doesn’t matter if some of the data is lost.

- **QoS 1: AT LEAST ONCE**
  
  Guarantees that a message will reach its intended recipient one or more times. The sender will continue to send the message until it receives an acknowledgment from the recipient, confirming it has received the message. The result of this QoS is that the recipient may receive the message multiple times, and also increases the network overhead than QoS 0, (due to acks).
In addition more burden is placed on the sender as it needs to store the message and retry should it fail to receive an ack in a reasonable time.

- **QoS 2: EXACTLY ONCE**

  The most costly of the QoS (in terms of network traffic and burden on sender and receiver) this QoS will ensure that the message is received by a recipient exactly one time. This ensures that the receiver never gets any duplicate copies of the message and will eventually get it, but at the extra cost of network overhead and complexity required on the sender and receiver.

### 8.2. MQTT Retain Messages

MQTT has an interesting feature in which messages can be "retained" for a particular address. This means that once a retain message has been sent to an address, any new subscribers to that address will receive the last sent retained message before any others messages. This happens even if the retained message was sent before a client has connected or subscribed. An example of where this feature might be useful is in environments such as IoT where devices need to quickly get the current state of a system when they are on boarded into a system.

Retained messages are stored in a queue named with a special prefix according to the name of the topic where they were originally sent. For example, a retained message sent to the topic `/abc/123` will be stored in a multicast queue named `$sys.mqtt.retain.abc.123` with an address of the same name. The MQTT specification doesn't define how long retained messages should be stored so the broker will hold on to this data until a client explicitly deletes the retained message or it potentially expires. However, even at that point the queue and address for the retained message will remain. These resources can be automatically deleted via the following `address-setting`:

```xml
<address-setting match="$sys.mqtt.retain.#">
    <auto-delete-queues>true</auto-delete-queues>
    <auto-delete-addresses>true</auto-delete-addresses>
</address-setting>
```

Keep in mind that it's also possible to automatically apply an `expiry-delay` to retained messages as well.

### 8.3. Will Messages

A will message can be sent when a client initially connects to a broker. Clients are able to set a "will message" as part of the connect packet. If the client abnormally disconnects, say due to a device or network failure the broker will proceed to publish the will message to the specified address (as defined also in the connect packet). Other subscribers to the will topic will receive the will message and can react accordingly. This feature can be useful in an IoT style scenario to detect errors across a potentially large scale deployment of devices.

### 8.4. Debug Logging

Detailed protocol logging (e.g. packets in/out) can be activated by turning on `TRACE` logging for
The MQTT specification doesn’t dictate the format of the payloads which clients publish. As far as the broker is concerned a payload is just an array of bytes. However, to facilitate logging the broker will encode the payloads as UTF-8 strings and print them up to 256 characters. Payload logging is limited to avoid filling the logs with potentially hundreds of megabytes of unhelpful information.

8.5. Persistent Subscriptions

The subscription information for MQTT sessions is stored in an internal queue named $sys.mqtt.sessions and persisted to storage (assuming persistence is enabled). The information is durable so that MQTT subscribers can reconnect and resume their subscriptions seamlessly after a broker restart, failure, etc. When brokers are configured for high availability this information will be available on the backup so even in the case of a broker fail-over subscribers will be able to resume their subscriptions.

While persistent subscriptions can be convenient they impose a performance penalty since data must be written to storage. If you don’t need the convenience (e.g. you always use clean sessions) and you don’t want the performance penalty then you can disable it by disabling durability for the $sys.mqtt.sessions queue in broker.xml, e.g.:

```xml
<addresses>
  ...
  <address name="$sys.mqtt.sessions">
    <anycast>
      <queue name="$sys.mqtt.sessions">
        <durable>false</durable>
      </queue>
    </anycast>
  </address>
  ...
</addresses>
```

The setting mqtt-session-state-persistence-timeout controls how long the broker will wait for the data to be written to storage before throwing an error. It is measured in milliseconds. The default is 5000.

8.6. Custom Client ID Handling

The client ID used by an MQTT application is very important as it uniquely identifies the application. In some situations broker administrators may want to perform extra validation or even modify incoming client IDs to support specific use-cases. This is possible by implementing a custom security manager as demonstrated in the security-manager example.

The simplest implementation is a “wrapper” just like the security-manager example uses. In the authenticate method you can modify the client ID using setClientId() on the
org.apache.activemq.artemis.spi.core.protocol.RemotingConnection that is passed in. If you perform some custom validation of the client ID you can reject the client ID by throwing a org.apache.activemq.artemis.core.protocol.mqtt.exceptions.InvalidClientIdException.

8.7. Wildcard subscriptions

MQTT defines a special wildcard syntax for topic filters. This definition is found in section 4.7.1 of both the 3.1.1 and 5 specs. MQTT topics are hierarchical much like a file system, and they use a special character (i.e. / by default) to separate hierarchical levels. Subscribers are able to subscribe to specific topics or to whole branches of a hierarchy.

To subscribe to branches of an address hierarchy a subscriber can use wild cards. There are 2 types of wildcards in MQTT:

- **Multi level (#)**

  Adding this wildcard to an address would match all branches of the address hierarchy under a specified node. For example: /uk/ Would match /uk/cities, /uk/cities/newcastle and also /uk/rivers/tyne. Subscribing to an address would result in subscribing to all topics in the broker. This can be useful, but should be done so with care since it has significant performance implications.

- **Single level (+)**

  Matches a single level in the address hierarchy. For example /uk/+stores would match /uk/newcastle/stores but not /uk/cities/newcastle/stores.

This is close to the default wildcard syntax, but not exactly the same. Therefore, some conversion is necessary. This conversion isn't free so if you want the best MQTT performance use broker.xml to configure the wildcard syntax to match MQTT's, e.g.:

```xml
<core>
  ...
  <wildcard-addresses>
    <delimiter>\</delimiter>
    <any-words>#</any-words>
    <single-word>+</single-word>
  </wildcard-addresses>
  ...
</core>
```

Of course, changing the default syntax also means other clients on other protocols will need to follow this same syntax as well as the match values of your address-setting configuration elements.

8.8. Web Sockets

Apache ActiveMQ Artemis also supports MQTT over Web Sockets. Modern web browsers which support Web Sockets can send and receive MQTT messages.
MQTT over Web Sockets is supported via a normal MQTT acceptor:

```
<acceptor name="mqtt-ws-acceptor">tcp://host:1883?protocols=MQTT</acceptor>
```

With this configuration, Apache ActiveMQ Artemis will accept MQTT connections over Web Sockets on the port 1883. Web browsers can then connect to `ws://<server>:1883` using a Web Socket to send and receive MQTT messages.

SSL/TLS is also available, e.g.:

```
<acceptor name="mqtt-wss-acceptor">
tcp://host:8883?protocols=MQTT;sslEnabled=true;keyStorePath=/path/to/keystore;keyStorePassword=myPass
</acceptor>
```

Web browsers can then connect to `wss://<server>:8883` using a Web Socket to send and receive MQTT messages.

### 8.9. Link Stealing

The MQTT specifications define a behavior often referred to as "link stealing." This means that whenever a new client connects with the same client ID as another existing client then the existing client's session will be closed and its network connection will be terminated.

In certain use-cases this behavior is not desired so it is configurable. The URL parameter `allowLinkStealing` can be configured on the MQTT acceptor to modify this behavior. By default `allowLinkStealing` is `true`. If it is set to `false` then whenever a new client connects with the same client ID as another existing client then the new client's session will be closed and its network connection will be terminated. In the case of MQTT 5 clients they will receive a disconnect reason code of `0x80` (i.e. "Unspecified error").

### 8.10. Automatic Subscription Clean-up

Sometimes MQTT 3.x clients using `CleanSession=false` don't properly unsubscribe. The URL parameter `defaultMqttSessionExpiryInterval` can be configured on the MQTT acceptor so that abandoned sessions and subscription queues will be cleaned up automatically after the expiry interval elapses.

MQTT 5 has the same basic semantics with slightly different configuration. The `CleanSession` flag was replaced with `CleanStart` and a `session expiry interval` property. The broker will use the client's session expiry interval if it is set. If it is not set then the broker will apply the `defaultMqttSessionExpiryInterval`.

The default `defaultMqttSessionExpiryInterval` is `-1` which means no clean up will happen for MQTT 3.x clients or for MQTT 5 clients which do not pass their own session expiry interval. Otherwise it represents the number of `seconds` which must elapse after the client has disconnected before the broker will remove the session state and subscription queues.
MQTT session state is scanned every 5,000 milliseconds by default. This can be changed using the \texttt{mqtt-session-scan-interval} element set in the \texttt{core} section of \texttt{broker.xml}.

### 8.11. Flow Control

MQTT 5 introduced a simple form of flow control. In short, a broker can tell a client how many QoS 1 & 2 messages it can receive before being acknowledged and vice versa.

This is controlled on the broker by setting the \texttt{receiveMaximum} URL parameter on the MQTT acceptor in \texttt{broker.xml}.

The default value is \texttt{65535} (the maximum value of the 2-byte integer used by MQTT).

A value of \texttt{0} is prohibited by the MQTT 5 specification.

A value of \texttt{-1} will prevent the broker from informing the client of any receive maximum which means flow-control will be disabled from clients to the broker. This is effectively the same as setting the value to \texttt{65535}, but reduces the size of the \texttt{CONNACK} packet by a few bytes.

### 8.12. Topic Alias Maximum

MQTT 5 introduced \hspace{1em} https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html#Topic_Alias[topic aliasing]. This is an optimization for the size of \texttt{PUBLISH} control packets as a 2-byte integer value can now be substituted for the _name of the topic which can potentially be quite long.

Both the client and the broker can inform each other about the maximum alias value they support (i.e. how many different aliases they support). This is controlled on the broker using the \texttt{topicAliasMaximum} URL parameter on the acceptor used by the MQTT client.

The default value is \texttt{65535} (the maximum value of the 2-byte integer used by MQTT).

A value of \texttt{0} will disable topic aliasing from clients to the broker.

A value of \texttt{-1} will prevent the broker from informing the client of any topic alias maximum which means aliasing will be disabled from clients to the broker. This is effectively the same as setting the value to \texttt{0}, but reduces the size of the \texttt{CONNACK} packet by a few bytes.

### 8.13. Maximum Packet Size

MQTT 5 introduced the maximum packet size. This is the maximum packet size the server or client is willing to accept.

This is controlled on the broker by setting the \texttt{maximumPacketSize} URL parameter on the MQTT acceptor in \texttt{broker.xml}.

The default value is \texttt{268435455} (i.e. 256MB - the maximum value of the variable byte integer used by MQTT).
A value of 0 is prohibited by the MQTT 5 specification.

A value of -1 will prevent the broker from informing the client of any maximum packet size which means no limit will be enforced on the size of incoming packets. This also reduces the size of the CONNACK packet by a few bytes.


All MQTT versions support a connection keep alive value defined by the client. MQTT 5 introduced a server keep alive value so that a broker can define the value that the client should use. The primary use of the server keep alive is for the server to inform the client that it will disconnect the client for inactivity sooner than the keep alive specified by the client.

This is controlled on the broker by setting the serverKeepAlive URL parameter on the MQTT acceptor in broker.xml.

The default value is 60 and is measured in seconds.

A value of 0 completely disables keep alives no matter the client's keep alive value. This is not recommended because disabling keep alives is generally considered dangerous since it could lead to resource exhaustion.

A value of -1 means the broker will always accept the client's keep alive value (even if that value is 0).

Any other value means the serverKeepAlive will be applied if it is less than the client's keep alive value unless the client's keep alive value is 0 in which case the serverKeepAlive is applied. This is because a value of 0 would disable keep alives and disabling keep alives is generally considered dangerous since it could lead to resource exhaustion.

8.15. Enhanced Authentication

MQTT 5 introduced enhanced authentication which extends the existing name & password authentication to include challenge / response style authentication.

However, there are currently no challenge / response mechanisms implemented so if a client passes the "Authentication Method" property in its CONNECT packet it will receive a CONNACK with a reason code of 0x8C (i.e. bad authentication method) and the network connection will be closed.

8.16. Publish Authorization Failures

The MQTT 3.1.1 specification is ambiguous regarding the broker's behavior when a PUBLISH packet fails due to a lack of authorization. In section 3.3.5 it says:

If a Server implementation does not authorize a PUBLISH to be performed by a Client; it has no way of informing that Client. It MUST either make a positive acknowledgement, according to the normal QoS rules, or close the
Network Connection

By default the broker will close the network connection. However if you'd rather have the broker make a positive acknowledgement then set the URL parameter `closeMqttConnectionOnPublishAuthorizationFailure` to `false` on the relevant MQTT acceptor in `broker.xml`, e.g.:

```xml
<acceptor name="mqtt">
tcp://0.0.0:1883?protocols=MQTT;closeMqttConnectionOnPublishAuthorizationFailure=false
</acceptor>
```
Apache ActiveMQ Artemis supports the OpenWire protocol so that an Apache ActiveMQ "Classic" JMS client can talk directly to an Apache ActiveMQ Artemis server. By default there is an acceptor configured to accept OpenWire connections on port 61616.

See the general Protocols and Interoperability chapter for details on configuring an acceptor for OpenWire.

Refer to the OpenWire examples for a look at this functionality in action.

9.1. Connection Monitoring

OpenWire has a few parameters to control how each connection is monitored, they are:

maxInactivityDuration
   It specifies the time (milliseconds) after which the connection is closed by the broker if no data was received. Default value is 30000.

maxInactivityDurationInitialDelay
   It specifies the maximum delay (milliseconds) before inactivity monitoring is started on the connection. It can be useful if a broker is under load with many connections being created concurrently. Default value is 10000.

useInactivityMonitor
   A value of false disables the InactivityMonitor completely and connections will never time out. By default it is enabled. On broker side you don't neet set this. Instead you can set the connection-ttl to -1.

useKeepAlive
   Indicates whether to send a KeepAliveInfo on an idle connection to prevent it from timing out. Enabled by default. Disabling the keep alive will still make connections time out if no data was received on the connection for the specified amount of time.

Note at the beginning the InactivityMonitor negotiates the appropriate maxInactivityDuration and maxInactivityDurationInitialDelay. The shortest duration is taken for the connection.

Fore more details please see ActiveMQ InactivityMonitor.

9.2. Disable/Enable Advisories

By default, advisory topics (ActiveMQ Advisory) are created in order to send certain type of advisory messages to listening clients. As a result, advisory addresses and queues will be displayed on the management console, along with user deployed addresses and queues. This sometimes cause confusion because the advisory objects are internally managed without user being aware of them. In addition, users may not want the advisory topics at all (they cause extra resources and performance penalty) and it is convenient to disable them at all from the broker side.
The protocol provides two parameters to control advisory behaviors on the broker side.

**supportAdvisory**
Indicates whether the broker supports advisory messages. If the value is true, advisory addresses/queues will be created. If the value is false, no advisory addresses/queues are created. Default value is true.

**suppressInternalManagementObjects**
Indicates whether advisory addresses/queues, if any, will be registered to management service (e.g. JMX registry). If set to true, no advisory addresses/queues will be registered. If set to false, those are registered and will be displayed on the management console. Default value is true.

The two parameters are configured on an OpenWire acceptor, e.g.:

```
<acceptor name="artemis">
tcp://localhost:61616?protocols=OPENWIRE;supportAdvisory=true;suppressInternalManagementObjects=false</acceptor>
```

### 9.3. OpenWire Destination Cache

For improved performance of the broker we keep a cache of recently used destinations, so that when new messages are dispatched to them, we do not have to do a lookup every time. By default, this cache holds up to 16 destinations. If additional destinations are added they will overwrite older records. If you are dealing with a large amount of queues you might want to increase this value, which is done via configuration option: `openWireDestinationCacheSize` set on the OpenWire acceptor like this:

```
<acceptor name="artemis">
tcp://localhost:61616?protocols=OPENWIRE;openWireDestinationCacheSize=64</acceptor>
```

This cache has to be set to a power of 2, i.e.: 2, 16, 128 and so on.

### 9.4. Virtual Topic Consumer Destination Translation

For existing OpenWire consumers of virtual topic destinations it is possible to configure a mapping function that will translate the virtual topic consumer destination into a FQQN address. This address will then represents the consumer as a multicast binding to an address representing the virtual topic.

The configuration string list property `virtualTopicConsumerWildcards` has parts separated by a ;. The first is the classic style destination filter that identifies the destination as belonging to a virtual topic. The second identifies the number of paths that identify the consumer queue such that it can be parsed from the destination. Any subsequent parts are additional configuration parameters for that mapping.

For example, the default virtual topic with consumer prefix of Consumer.., would require a
virtualTopicConsumerWildcards filter of Consumer..>;2. As a url parameter this transforms to Consumer.*.%;3B2 when the url significant characters >; are escaped with their hex code points. In an acceptor url it would be:

```
<acceptor name="artemis"
```

This will translate Consumer.A.VirtualTopic.Orders into a FQCN of VirtualTopic.Orders::Consumer.A.VirtualTopic.Orders using the int component 2 of the configuration to identify the consumer queue as the first two paths of the destination. virtualTopicConsumerWildcards is multi valued using a , separator.

### 9.4.1. selectorAware

The mappings support an optional parameter, selectorAware which when true, transfers any selector information from the OpenWire consumer into a queue filter of any auto-created subscription queue.

> the selector/filter is persisted with the queue binding in the normal way, such that it works independent of connected consumers.

Please see Virtual Topic Mapping example contained in the OpenWire examples.
Chapter 10. Using Core

Apache ActiveMQ Artemis core is a messaging system with its own API. We call this the core API.

If you don't want to use the JMS API or any of the other supported protocols you can use the core API directly. The core API provides all the functionality of JMS but without much of the complexity. It also provides features that are not available using JMS.

10.1. Core Messaging Concepts

Some of the core messaging concepts are similar to JMS concepts, but core messaging concepts are also different in some ways as well. In general the core API is simpler than the JMS API, since we remove distinctions between queues, topics and subscriptions. We'll discuss each of the major core messaging concepts in turn, but to see the API in detail please consult the Javadoc.

Also refer to the address model chapter for a high-level overview of these concepts as well as configuration details.

10.1.1. Message

- A message is the unit of data which is sent between clients and servers.
- A message has a body which is a buffer containing convenient methods for reading and writing data into it.
- A message has a set of properties which are key-value pairs. Each property key is a string and property values can be of type integer, long, short, byte, byte[], String, double, float or boolean.
- A message has an address it is being sent to. When the message arrives on the server it is routed to any queues that are bound to the address. The routing semantics (i.e. anycast or multicast) are determined by the “routing type” of the address and queue. If the queues are bound with any filter, the message will only be routed to that queue if the filter matches. An address may have many queues bound to it or even none. There may also be entities other than queues (e.g. diverts) bound to addresses.
- Messages can be either durable or non durable. Durable messages in a durable queue will survive a server crash or restart. Non durable messages will never survive a server crash or restart.
- Messages can be specified with a priority value between 0 and 9. 0 represents the lowest priority and 9 represents the highest. The broker will attempt to deliver higher priority messages before lower priority ones.
- Messages can be specified with an optional expiry time. The broker will not deliver messages after its expiry time has been exceeded.
- Messages also have an optional timestamp which represents the time the message was sent.
- Apache ActiveMQ Artemis also supports the sending/consuming of very large messages much larger than can fit in available RAM at any one time.
10.2. Core API

10.2.1. ServerLocator

Clients use ServerLocator instances to create ClientSessionFactory instances. ServerLocator instances are used to locate servers and create connections to them.

In JMS terms think of a ServerLocator in the same way you would a JMS Connection Factory.

ServerLocator instances are created using the ActiveMQClient factory class.

10.2.2. ClientSessionFactory

Clients use ClientSessionFactory instances to create ClientSession instances. ClientSessionFactory instances are basically the connection to a server.

In JMS terms think of them as JMS Connections.

ClientSessionFactory instances are created using the ServerLocator class.

10.2.3. ClientSession

A client uses a ClientSession for consuming and producing messages and for grouping them in transactions. ClientSession instances can support both transactional and non transactional semantics and also provide an XAResource interface so messaging operations can be performed as part of a JTA transaction.

ClientSession instances group ClientConsumer instances and ClientProducer instances.

ClientSession instances can be registered with an optional SendAcknowledgementHandler. This allows your client code to be notified asynchronously when sent messages have successfully reached the server. This unique Apache ActiveMQ Artemis feature, allows you to have full guarantees that sent messages have reached the server without having to block on each message sent until a response is received. Blocking on each messages sent is costly since it requires a network round trip for each message sent. By not blocking and receiving send acknowledgements asynchronously you can create true end to end asynchronous systems which is not possible using the standard JMS API. For more information on this advanced feature please see the section Guarantees of sends and commits.

Identifying your session for management and debugging

Assigning IDs to your core sessions can help you with monitoring and debugging the cluster using the management console.

```java
ClientSession session;
// ...
session.addMetaData(ClientSession.JMS_SESSION_IDENTIFIER_PROPERTY, "jms-client-id");
session.addMetaData("jms-client-id", "my-session");
```
Such ID will then appear in the Client ID column under the Connections, Consumers and Producers tabs.

If you are using the JMS API, the setClientID would give you the same effect.

### 10.2.4. ClientConsumer

Clients use ClientConsumer instances to consume messages from a queue. Core messaging supports both synchronous and asynchronous message consumption semantics. ClientConsumer instances can be configured with an optional filter expression and will only consume messages which match that expression.

### 10.2.5. ClientProducer

Clients create ClientProducer instances on ClientSession instances so they can send messages. ClientProducer instances can specify an address to which all sent messages are routed, or they can have no specified address, and the address is specified at send time for the message.

Please note that ClientSession, ClientProducer and ClientConsumer instances are designed to be re-used.

It’s an anti-pattern to create new ClientSession, ClientProducer and ClientConsumer instances for each message you produce or consume. If you do this, your application will perform very poorly. This is discussed further in the section on performance tuning Performance Tuning.

### 10.3. A simple example of using Core

Here’s a very simple program using the core messaging API to send and receive a message. Logically it’s comprised of two sections: firstly setting up the producer to write a message to an address, and secondly, creating a queue for the consumer using anycast routing, creating the consumer, and starting it.

```java
ServerLocator locator = ActiveMQClient.createServerLocator("vm://0");

// In this simple example, we just use one session for both producing and receiving
ClientSessionFactory factory = locator.createSessionFactory();
ClientSession session = factory.createSession();

// A producer is associated with an address ...
ClientProducer producer = session.createProducer("example");
ClientMessage message = session.createMessage(true);
message.getBodyBuffer().writeString("Hello");

// We need a queue attached to the address ...
```
session.createQueue("example", RoutingType.ANYCAST, "example", true);

// And a consumer attached to the queue ...
ClientConsumer consumer = session.createConsumer("example");

// Once we have a queue, we can send the message ...
producer.send(message);

// We need to start the session before we can -receive- messages ...
session.start();
ClientMessage msgReceived = consumer.receive();

System.out.println("message = " + msgReceived.getBodyBuffer().readString());
session.close();
Chapter 11. Core Client Failover

Apache ActiveMQ Artemis clients can be configured to automatically reconnect to the same server, reconnect to the backup server or reconnect to other active servers in the event that a failure is detected in the connection between the client and the server. The clients detect connection failure when they have not received any packets from the server within the time given by client-failure-check-period as explained in section Detecting Dead Connections.

11.1. Reconnect to the same server

Set reconnectAttempts to any non-zero value to reconnect to the same server, for further details see Reconnection and failover attributes.

If the disconnection was due to some transient failure such as a temporary network outage and the target server was not restarted, then the sessions will still exist on the server, assuming the client hasn’t been disconnected for more than connection-ttl

In this scenario, the client sessions will be automatically re-attached to the server sessions after the reconnection. This is done 100% transparently and the client can continue exactly as if nothing had happened.

The way this works is as follows:

As Apache ActiveMQ Artemis clients send commands to their servers they store each sent command in an in-memory buffer. In the case that connection failure occurs and the client subsequently reattaches to the same server, as part of the reattachment protocol the server informs the client during reattachment with the id of the last command it successfully received from that client.

If the client has sent more commands than were received before failover it can replay any sent commands from its buffer so that the client and server can reconcile their states. Ac

The size of this buffer is configured with the confirmationWindowSize parameter on the connection URL. When the server has received confirmationWindowSize bytes of commands and processed them it will send back a command confirmation to the client, and the client can then free up space in the buffer.

The window is specified in bytes.

Setting this parameter to -1 disables any buffering and prevents any re-attachment from occurring, forcing reconnect instead. The default value for this parameter is -1. (Which means by default no auto re-attachment will occur)

11.2. Reconnect to the backup server

Set reconnectAttempts to any non-zero value and ha to true to reconnect to the back server, for further details see Reconnection and failover attributes.

The clients can be configured to discover the list of live-backup server groups in a number of different ways. They can be configured explicitly or probably the most common way of doing this is...
to use server discovery for the client to automatically discover the list. For full details on how to configure server discovery, please see Clusters. Alternatively, the clients can explicitly connect to a specific server and download the current servers and backups see Clusters.

By default, failover will only occur after at least one connection has been made. In other words, by default, failover will not occur if the client fails to make an initial connection - in this case it will simply retry connecting according to the reconnect-attempts property and fail after this number of attempts.

11.3. Reconnect to other active servers

Set failoverAttempts to any non-zero value to reconnect to other active servers, for further details see Reconnection and failover attributes.

If reconnectAttempts value is not zero then the client will try to reconnect to other active servers only after all attempts to reconnect to the same server or reconnect to the backup server fail.

11.4. Session reconnection

When clients reconnect to the same server after a restart, reconnect to the backup server or reconnect to other active servers any sessions will no longer exist on the server and it won’t be possible to 100% transparently re-attach to them. In this case, any sessions and consumers on the client will be automatically recreated on the server.

Client reconnection is also used internally by components such as core bridges to allow them to reconnect to their target servers.

11.5. Failing over on the initial connection

Since the client does not learn about the full topology until after the first connection is made there is a window where it does not know about the backup. If a failure happens at this point the client can only try reconnecting to the original server. To configure how many attempts the client will make you can set the URL parameter initialConnectAttempts. The default for this is 0, that is try only once. Once the number of attempts has been made an exception will be thrown.

For examples of automatic failover with transacted and non-transacted JMS sessions, please see the examples chapter.

11.6. Reconnection and failover attributes

Client reconnection and failover is configured using the following parameters:

retryInterval

This optional parameter determines the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is 2000 milliseconds.
**retryIntervalMultiplier**

This optional parameter determines a multiplier to apply to the time since the last retry to compute the time to the next retry.

This allows you to implement an *exponential backoff* between retry attempts.

Let's take an example:

If we set `retryInterval` to 1000 ms and we set `retryIntervalMultiplier` to 2.0, then, if the first reconnect attempt fails, we will wait 1000 ms then 2000 ms then 4000 ms between subsequent reconnection attempts.

The default value is 1.0 meaning each reconnect attempt is spaced at equal intervals.

**maxRetryInterval**

This optional parameter determines the maximum retry interval that will be used. When setting `retryIntervalMultiplier` it would otherwise be possible that subsequent retries exponentially increase to ridiculously large values. By setting this parameter you can set an upper limit on that value. The default value is 2000 milliseconds.

**ha**

This optional parameter determines whether the client will try to reconnect to the backup node when the primary node is not reachable. The default value is false. For more information on HA, please see High Availability and Failover.

**reconnectAttempts**

This optional parameter determines the total number of reconnect attempts to make to the current live/backup pair before giving up. A value of -1 signifies an unlimited number of attempts. The default value is 0.

**failoverAttempts**

This optional parameter determines the total number of failover attempts to make after a reconnection failure before giving up and shutting down. A value of -1 signifies an unlimited number of attempts. The default value is 0.

All of these parameters are set on the URL used to connect to the broker.

If your client does manage to reconnect but the session is no longer available on the server, for instance if the server has been restarted or it has timed out, then the client won't be able to re-attach, and any ExceptionListener or FailureListener instances registered on the connection or session will be called.

### 11.7. ExceptionListeners and SessionFailureListeners

Please note, that when a client reconnects or re-attaches, any registered JMS ExceptionListener or core API SessionFailureListener will be called.
Chapter 12. Mapping JMS Concepts to the Core API

This chapter describes how JMS destinations are mapped to the Apache ActiveMQ Artemis address model. If you haven’t already done so, please read about the address model as it introduces concepts which are referenced here.

12.1. JMS Topic

A JMS topic is implemented as a address where the name of the address is the same as the name of the JMS topic.

A subscription on that JMS topic is represented as a multicast queue on the corresponding address. The queue is named according to the whether the subscription is durable and according to the client ID and subscription named provided via the JMS API.

Typically there is just one consumer per queue, but there can be multiple consumers on a queue when using JMS shared topic subscriptions. Any messages sent to the JMS topic is therefore routed to every multicast queue bound to the corresponding address and then dispatched to any consumers on those queues (i.e. JMS topic subscriber). If there are no queues on the address then the message is simply dropped.

This effectively achieves JMS pub/sub semantics.

12.2. JMS Queue

Likewise, a JMS queue is implemented as an address where the name of the address is the same as the name of the JMS queue.

However, there will be just one anycast queue on the corresponding address. All JMS consumers on this queue will share the messages in the queue. The queue is named the same as the address.

This effectively achieves JMS point-to-point semantics.
Chapter 13. Using JMS or Jakarta Messaging

Although Apache ActiveMQ Artemis provides a JMS agnostic messaging API, many users will be more comfortable using JMS.

JMS is a very popular API standard for messaging, and most messaging systems provide a JMS API. If you are completely new to JMS we suggest you follow the Oracle JMS tutorial - a full JMS tutorial is out of scope for this guide.

Apache ActiveMQ Artemis also ships with a wide range of examples, many of which demonstrate JMS API usage. A good place to start would be to play around with the simple JMS Queue and Topic example, but we also provide examples for many other parts of the JMS API. A full description of the examples is available in Examples.

In this section we’ll go through the main steps in configuring the server for JMS and creating a simple JMS program. We’ll also show how to configure and use JNDI, and also how to use JMS with Apache ActiveMQ Artemis without using any JNDI.

13.1. A simple ordering system

For this chapter we’re going to use a very simple ordering system as our example. It is a somewhat contrived example because of its extreme simplicity, but it serves to demonstrate the very basics of setting up and using JMS.

We will have a single JMS Queue called OrderQueue, and we will have a single MessageProducer sending an order message to the queue and a single MessageConsumer consuming the order message from the queue.

The queue will be a durable queue, i.e. it will survive a server restart or crash. We also want to pre-deploy the queue, i.e. specify the queue in the server configuration so it is created automatically without us having to explicitly create it from the client.

13.2. JNDI

The JMS specification establishes the convention that administered objects (i.e. JMS queue, topic and connection factory instances) are made available via the JNDI API. Brokers are free to implement JNDI as they see fit assuming the implementation fits the API. Apache ActiveMQ Artemis does not have a JNDI server. Rather, it uses a client-side JNDI implementation that relies on special properties set in the environment to construct the appropriate JMS objects. In other words, no objects are stored in JNDI on the Apache ActiveMQ Artemis server, instead they are simply instantiated on the client based on the provided configuration. Let’s look at the different kinds of administered objects and how to configure them.

The following configuration properties are strictly required when Apache ActiveMQ Artemis is running in stand-alone mode. When Apache ActiveMQ Artemis is integrated to an application server (e.g. Wildfly) the application server itself will almost certainly provide a JNDI client with its own properties.
13.2.1. ConnectionFactory JNDI

A JMS connection factory is used by the client to make connections to the server. It knows the location of the server it is connecting to, as well as many other configuration parameters.

Here's a simple example of the JNDI context environment for a client looking up a connection factory to access an embedded instance of Apache ActiveMQ Artemis:

```java
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.invmConnectionFactory=vm://0
```

In this instance we have created a connection factory that is bound to `invmConnectionFactory`, any entry with prefix `connectionFactory` will create a connection factory.

In certain situations there could be multiple server instances running within a particular JVM. In that situation each server would typically have an InVM acceptor with a unique server-ID. A client using JMS and JNDI can account for this by specifying a connection factory for each server, like so:

```java
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.invmConnectionFactory0=vm://0
connectionFactory.invmConnectionFactory1=vm://1
connectionFactory.invmConnectionFactory2=vm://2
```

Here is a list of all the supported URL schemes:

- `vm`
- `tcp`
- `udp`
- `jgroups`

Most clients won’t be connecting to an embedded broker. Clients will most commonly connect across a network a remote broker. Here's a simple example of a client configuring a connection factory to connect to a remote broker running on `myhost:5445`:

```java
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.ConnectionFactory=tcp://myhost:5445
```

In the example above the client is using the `tcp` scheme for the provider URL. A client may also specify multiple comma-delimited host:port combinations in the URL (e.g. `(tcp://remote-host1:5445,remote-host2:5445)`). Whether there is one or many host:port combinations in the URL they are treated as the `initial connector(s)` for the underlying connection.

The `udp` scheme is also supported which should use a host:port combination that matches the group-
address and group-port from the corresponding broadcast-group configured on the ActiveMQ Artemis server(s).

Each scheme has a specific set of properties which can be set using the traditional URL query string format (e.g. scheme://host:port?key1=value1&key2=value2) to customize the underlying transport mechanism. For example, if a client wanted to connect to a remote server using TCP and SSL it would create a connection factory like so, tcp://remote-host:5445?ssl-enabled=true.

All the properties available for the tcp scheme are described in the documentation regarding the Netty transport.

Note if you are using the tcp scheme and multiple addresses then a query can be applied to all the url's or just to an individual connector, so where you have

- (tcp://remote-host1:5445?httpEnabled=true,remote-host2:5445?httpEnabled=true)?clientId=1234

then the httpEnabled property is only set on the individual connectors where as the clientId is set on the actual connection factory. Any connector specific properties set on the whole URI will be applied to all the connectors.

The udp scheme supports 4 properties:

**localAddress**

If you are running with multiple network interfaces on the same machine, you may want to specify that the discovery group listens only on a specific interface. To do this you can specify the interface address with this parameter.

**localPort**

If you want to specify a local port to which the datagram socket is bound you can specify it here. Normally you would just use the default value of -1 which signifies that an anonymous port should be used. This parameter is always specified in conjunction with localAddress.

**refreshTimeout**

This is the period the discovery group waits after receiving the last broadcast from a particular server before removing that servers connector pair entry from its list. You would normally set this to a value significantly higher than the broadcast-period on the broadcast group otherwise servers might intermittently disappear from the list even though they are still broadcasting due to slight differences in timing. This parameter is optional, the default value is 10000 milliseconds (10 seconds).

**discoveryInitialWaitTimeout**

If the connection factory is used immediately after creation then it may not have had enough time to received broadcasts from all the nodes in the cluster. On first usage, the connection factory will make sure it waits this long since creation before creating the first connection. The default value for this parameter is 10000 milliseconds.

Lastly, the jgroups scheme is supported which provides an alternative to the udp scheme for server discovery. The URL pattern is jgroups://channelName?file=jgroups-xml-conf-filename where jgroups-xml-conf-filename refers to an XML file on the classpath that contains the JGroups configuration.
The `channelName` is the name given to the jgroups channel created.

The `refreshTimeout` and `discoveryInitialWaitTimeout` properties are supported just like with `udp`.

The default type for the default connection factory is of type `javax.jms.ConnectionFactory` or `jakarta.jms.ConnectionFactory` depending on the client you’re using. This can be changed by setting the type like so

```java
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
java.naming.provider.url=tcp://localhost:5445?type=CF
```

In this example it is still set to the default, below shows a list of types that can be set.

**Configuration for Connection Factory Types**

The interface provided will depend on whether you’re using the JMS or Jakarta Messaging client implementation.

<table>
<thead>
<tr>
<th>type</th>
<th>interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF (default)</td>
<td><code>javax.jms.ConnectionFactory</code> or <code>jakarta.jms.ConnectionFactory</code></td>
</tr>
<tr>
<td>XA_CF</td>
<td><code>javax.jms.XAConnectionFactory</code> or <code>jakarta.jms.XAConnectionFactory</code></td>
</tr>
<tr>
<td>QUEUE_CF</td>
<td><code>javax.jms.QueueConnectionFactory</code> or <code>jakarta.jms.QueueConnectionFactory</code></td>
</tr>
<tr>
<td>QUEUE_XA_CF</td>
<td><code>javax.jms.XAQueueConnectionFactory</code> or <code>jakarta.jms.XAQueueConnectionFactory</code></td>
</tr>
<tr>
<td>TOPIC_CF</td>
<td><code>javax.jms.TopicConnectionFactory</code> or <code>jakarta.jms.TopicConnectionFactory</code></td>
</tr>
<tr>
<td>TOPIC_XA_CF</td>
<td><code>javax.jms.XATopicConnectionFactory</code> or <code>jakarta.jms.XATopicConnectionFactory</code></td>
</tr>
</tbody>
</table>

**13.2.2. Destination JNDI**

JMS destinations are also typically looked up via JNDI. As with connection factories, destinations can be configured using special properties in the JNDI context environment. The property `name` should follow the pattern: `queue.<jndi-binding>` or `topic.<jndi-binding>`. The property `value` should be the name of the queue hosted by the Apache ActiveMQ Artemis server. For example, if the server had a JMS queue configured like so:

```xml
<address name="OrderQueue">
  <queue name="OrderQueue"/>
</address>
```

And if the client wanted to bind this queue to "queues/OrderQueue" then the JNDI properties would
be configured like so:

```java
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
java.naming.provider.url=tcp://myhost:5445
queue.queues/OrderQueue=OrderQueue
```

It is also possible to look-up JMS destinations which haven’t been configured explicitly in the JNDI context environment. This is possible using `dynamicQueues/` or `dynamicTopics/` in the look-up string. For example, if the client wanted to look-up the aforementioned "OrderQueue" it could do so simply by using the string "dynamicQueues/OrderQueue". Note, the text that follows `dynamicQueues/` or `dynamicTopics/` must correspond exactly to the name of the destination on the server.

### 13.2.3. The code

Here’s the code for the example:

First we’ll create a JNDI initial context from which to lookup our JMS objects. If the above properties are set in `jndi.properties` and it is on the classpath then any new, empty `InitialContext` will be initialized using those properties:

```java
InitialContext ic = new InitialContext();

//Now we'll look up the connection factory from which we can create
//connections to myhost:5445:
ConnectionFactory cf = (ConnectionFactory)ic.lookup("ConnectionFactory");

//And look up the Queue:
Queue orderQueue = (Queue)ic.lookup("queues/OrderQueue");

//Next we create a JMS connection using the connection factory:
Connection connection = cf.createConnection();

//And we create a non transacted JMS Session, with AUTO_ACKNOWLe.g. //acknowledge mode:
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

//We create a MessageProducer that will send orders to the queue:
MessageProducer producer = session.createProducer(orderQueue);

//And we create a MessageConsumer which will consume orders from the
//queue:
MessageConsumer consumer = session.createConsumer(orderQueue);
```
//We make sure we start the connection, or delivery won't occur on it:

connection.start();

//We create a simple TextMessage and send it:

TextMessage message = session.createTextMessage("This is an order");
producer.send(message);

//And we consume the message:

TextMessage receivedMessage = (TextMessage)consumer.receive();
System.out.println("Got order: " + receivedMessage.getText());

It is as simple as that. For a wide range of working JMS examples please see the [examples](#).

**Warning**

Please note that JMS connections, sessions, producers and consumers are *designed to be re-used*.

It is an anti-pattern to create new connections, sessions, producers and consumers for each message you produce or consume. If you do this, your application will perform very poorly. This is discussed further in the section on performance tuning [Performance Tuning](#).

### 13.3. Directly instantiating JMS Resources without using JNDI

Although it is a very common JMS usage pattern to lookup JMS *Administered Objects* (that's JMS Queue, Topic and ConnectionFactory instances) from JNDI, in some cases you just think "Why do I need JNDI? Why can't I just instantiate these objects directly?"

With Apache ActiveMQ Artemis you can do exactly that. Apache ActiveMQ Artemis supports the direct instantiation of JMS Queue, Topic and ConnectionFactory instances, so you don't have to use JNDI at all.

For a full working example of direct instantiation please look at the [Instantiate JMS Objects Directly](#) example under the JMS section of the examples.

Here's our simple example, rewritten to not use JNDI at all:

We create the JMS ConnectionFactory object via the ActiveMQJMSClient Utility class, note we need to provide connection parameters and specify which transport we are using, for more information
TransportConfiguration transportConfiguration = new TransportConfiguration (NettyConnectorFactory.class.getName());

ConnectionFactory cf = ActiveMQJMSClient.createConnectionFactoryWithoutHA (JMSFactoryType.CF,transportConfiguration);

//We also create the JMS Queue object via the ActiveMQJMSClient Utility class:
Queue orderQueue = ActiveMQJMSClient.createQueue("OrderQueue");

//Next we create a JMS connection using the connection factory:
Connection connection = cf.createConnection();

//And we create a non transacted JMS Session, with AUTO_ACKNOWLEDGE mode:
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

//We create a MessageProducer that will send orders to the queue:
MessageProducer producer = session.createProducer(orderQueue);

//And we create a MessageConsumer which will consume orders from the queue:
MessageConsumer consumer = session.createConsumer(orderQueue);

//We make sure we start the connection, or delivery won't occur on it:
connection.start();

//We create a simple TextMessage and send it:
TextMessage message = session.createTextMessage("This is an order");
producer.send(message);

//And we consume the message:
TextMessage receivedMessage = (TextMessage)consumer.receive();
System.out.println("Got order: " + receivedMessage.getText());

13.4. Setting The Client ID

This represents the client id for a JMS client and is needed for creating durable subscriptions. It is possible to configure this on the connection factory and can be set via the clientId element. Any
connection created by this connection factory will have this set as its client id.

13.5. Setting The Batch Size for DUPS_OK

When the JMS acknowledge mode is set to DUPS_OK it is possible to configure the consumer so that it sends acknowledgements in batches rather than one at a time, saving valuable bandwidth. This can be configured via the connection factory via the dupsOkBatchSize element and is set in bytes. The default is 1024 * 1024 bytes = 1 MiB.

13.6. Setting The Transaction Batch Size

When receiving messages in a transaction it is possible to configure the consumer to send acknowledgements in batches rather than individually saving valuable bandwidth. This can be configured on the connection factory via the transactionBatchSize element and is set in bytes. The default is 1024 * 1024.

13.7. Setting The Destination Cache

Many frameworks such as Spring resolve the destination by name on every operation, this can cause a performance issue and extra calls to the broker, in a scenario where destinations (addresses) are permanent broker side, such as they are managed by a platform or operations team. Using cacheDestinations element, you can toggle on the destination cache to improve the performance and reduce the calls to the broker. This should not be used if destinations (addresses) are not permanent broker side, as in dynamic creation/deletion.
Chapter 14. Extra Acknowledge Modes

JMS specifies 3 acknowledgement modes:

- AUTO_ACKNOWLEDGE
- CLIENT_ACKNOWLEDGE
- DUPS_OK_ACKNOWLEDGE

Apache ActiveMQ Artemis supports two additional modes: PRE_ACKNOWLEDGE and INDIVIDUAL_ACKNOWLEDGE

In some cases you can afford to lose messages in event of failure, so it would make sense to acknowledge the message on the server before delivering it to the client.

This extra mode is supported by Apache ActiveMQ Artemis and will call it pre-acknowledge mode.

The disadvantage of acknowledging on the server before delivery is that the message will be lost if the system crashes after acknowledging the message on the server but before it is delivered to the client. In that case, the message is lost and will not be recovered when the system restart.

Depending on your messaging case, preAcknowledgement mode can avoid extra network traffic and CPU at the cost of coping with message loss.

An example of a use case for pre-acknowledgement is for stock price update messages. With these messages it might be reasonable to lose a message in event of crash, since the next price update message will arrive soon, overriding the previous price.

Please note, that if you use pre-acknowledge mode, then you will lose transactional semantics for messages being consumed, since clearly they are being acknowledged first on the server, not when you commit the transaction. This may be stating the obvious but we like to be clear on these things to avoid confusion!

14.1. Using PRE_ACKNOWLEDGE

This can be configured by setting the boolean URL parameter preAcknowledge to true.

Alternatively, when using the JMS API, create a JMS Session with the ActiveMQSession.PRE_ACKNOWLEDGE constant.

```java
// messages will be acknowledge on the server *before* being delivered to the client
Session session = connection.createSession(false, ActiveMQJMSConstants.PRE_ACKNOWLEDGE);
```

14.2. Individual Acknowledge

A valid use-case for individual acknowledgement would be when you need to have your own scheduling and you don’t know when your message processing will be finished. You should prefer
having one consumer per thread worker but this is not possible in some circumstances depending on how complex is your processing. For that you can use the individual acknowledgement.

You basically setup Individual ACK by creating a session with the acknowledge mode with `ActiveMQConstants.INDIVIDUAL_ACKNOWLEDGE`. Individual ACK inherits all the semantics from Client Acknowledge, with the exception the message is individually acked.

Please note, that to avoid confusion on MDB processing, Individual ACKNOWLEDGE is not supported through MDBs (or the inbound resource adapter). this is because you have to finish the process of your message inside the MDB.

### 14.3. Example

See the Pre-acknowledge Example which shows how to use pre-acknowledgement mode with JMS.
Chapter 15. Versions

This chapter provides the following information for each release:

- A link to the full release notes which includes all issues resolved in the release.
- A brief list of “highlights” when applicable.
- If necessary, specific steps required when upgrading from the previous version.

If the upgrade spans multiple versions then the steps from each version need to be followed in order.

Follow the general upgrade procedure outlined in the Upgrading the Broker chapter in addition to any version-specific upgrade instructions outlined here.

15.1. 2.34.0

Full release notes

15.1.1. Highlights

- Extensive resiliency tests and hardening on Mirroring.
- Paging performance improvements on sync.
- Statistics about security events.
- Replication status metrics.

15.1.2. Upgrading from 2.33.0

- Due to ARTEMIS-4712 the connection pooling functionality configured via the connectionPool property in login.config is no longer supported in the LDAPLoginModule. The login.config may still use the connectionPool property. No error will be thrown. However, connections will no longer be pooled regardless of the configuration.

- Due to ARTEMIS-4498 the web console will now report all internal objects.
  - This was done in an attempt to allow administrators to act when things are not working as expected, to get metrics on these objects and allow more transparency for the broker.
  - this includes all Openwire Advisor queues and addresses, MQTT internal objects, Cluster Store and Forward (SNF) Queues, Mirror SNF.
  - You may want to revisit authorizations if you mean to control access to certain users on the web console.

- The CLI operation ./artemis queue stat has its output improved and updated. If you parsed the previous output in scripts you will see differences in the output.
  - It is not recommended to parse the output of a CLI Operation. You may use jolokia calls over management instead with proper JSON output.
15.2. 2.33.0

Full release notes

15.2.1. Highlights

- Support for JSON formatted typed properties on CLI `producer` command
- New CLI command `pwd` for showing directories related to the current instance
- Maven Bill of Materials (BOM) `artemis-bom` to simplify integration
- "FirstMessage" API for scheduled messages
- New "view" and "edit" permissions for management operations configurable via `security-settings` in `broker.xml`
- New `sslAutoReload` parameter for the embedded web server configured in `bootstrap.xml` to detect and automatically reload when SSL stores change on disk
- Performance improvements on mirroring and paging
- Logging metrics to mitigate the risk of missing `WARN` or `ERROR` messages in the log.
- Much improved documentation on network isolation (aka split brain)
- Pluggable lock manager (aka pluggable quorum voting) out of "experimental" status and ready for general use

15.2.2. Upgrading from 2.32.0

- Due to ARTEMIS-4532 the names of addresses and queues related to MQTT topics and subscriptions respectively may change. This will impact MQTT use-cases if both of the following are true:
  
  1. The broker is configured to use a wildcard syntax which *doesn’t match* the MQTT wildcard syntax (e.g. the default wildcard syntax).
  2. You are using characters from the broker’s wildcard syntax in your MQTT topic name or filter. For example, if you were using the default wildcard syntax and an MQTT topic named `1.0/group/device`. The dot (.) character here is part of the broker’s wildcard syntax, and it is being used in the name of an MQTT topic.

    In this case the characters from the broker’s wildcard syntax that do not match the characters in the MQTT wildcard syntax will be escaped with a backslash (i.e. \\). To avoid this conversion you can configure the broker to use the MQTT wildcard syntax or change the name of the MQTT topic name or filter.

    This change will also impact OpenWire JMS consumers which are using # instead of < for wildcard purposes. In previous versions the # character was just passed through when converting from the OpenWire wildcard format to the Core wildcard format. However, now the # character is escaped during conversion. It is a bug for an application to use to use # as a wildcard with the OpenWire JMS client; > is the proper character to use as specified in the ActiveMQ Classic documentation on wildcards.
- Due to ARTEMIS-4559 folks embedding the broker and also depending on the artemis-quorum-ri and/or artemis-quorum-api modules and/or using org.apache.activemq.artemis.core.config.ha.DistributedPrimitiveManagerConfiguration will need to use artemis-lockmanager-ri, artemis-lockmanager-api, and org.apache.activemq.artemis.core.config.ha.DistributedLockManagerConfiguration respectively. Previously these were marked as "experimental" in the documentation and were changed strictly in name to clarify their use conceptually. Furthermore, the documentation around high availability and network isolation (i.e. split brain) was refactored significantly to be more clear and comprehensive.

15.3. 2.32.0

Full release notes

15.3.1. Highlights

- Mirrored Core Messages can now be sent on their native format without conversions
- Mirror bug fixes and improvements
- ActiveMQ Artemis has now adopted more inclusive language definitions.
- The examples are now part of its own repository: https://github.com/apache/activemq-artemis-examples/

15.3.2. Upgrading from 2.31.x

- Due to ARTEMIS-4501 MQTT subscription queues will be automatically removed when the corresponding session expires, either based on the session expiry interval passed by an MQTT 5 client or based on the configured defaultMqttSessionExpiryInterval for MQTT 3.x clients or MQTT 5 clients which don't explicitly pass a session expiry interval.

Prior to this change removing subscription queues relied on the generic auto-delete-* address-settings.

These settings are now no longer required.

Configure defaultMqttSessionExpiryInterval instead.

- Due to ARTEMIS-3474 the following configuration elements have changed wherever they occur (e.g. broker.xml, bootstrap.xml, etc.), although all the previous configurations will still be supported for the time being:

  ◦ master → primary
  ◦ slave → backup
  ◦ check-for-live-server → check-for-active-server
  ◦ whitelist → allowlist
  ◦ blacklist → denylist

Additionally, references to these elements have also changed in the documentation and in
management interfaces. Cluster topology information (e.g. returned from the listNetworkTopology) will contain both primary and live entries for nodes functioning as primary servers.

15.4. 2.31.2

Full release notes

15.4.1. Highlights

• Bug Fix

15.5. 2.31.1

Full release notes

15.5.1. Highlights

• Bug Fixes and component upgrades

15.6. 2.31.0

Full release notes

15.6.1. Highlights

• Introduced an interactive shell for running CLI command as well as Bash & ZSH auto-complete support.

• Added a CLI cluster verification tool to help monitor broker topologies. Use via the check cluster command.

• The queue stat command is now able to to verify the message counts on the entire cluster topology when clustering is in use.

• Added AMQP Federation support to broker connections.

• MQTT subscription state is now persisted.

• Significantly improved the Paging JDBC Persistence.

• Converted much of the documentation from MarkDown to AsciiDoc. See ARTEMIS-4383 for more details.

• Many other bug fixes and improvements.

15.6.2. Upgrading from 2.30.0

• Due to ARTEMIS-4372 and the introduction of the new Artemis shell feature when you invoke ./artemis it will now start the new shell to navigate through the CLI commands rather than just spitting out the help text.
15.7. 2.30.0

Full release notes

15.7.1. Highlights

• This is mainly a bug-fix release with a few small improvements and a handful of dependency upgrades. See the release notes for all the details.

15.8. 2.29.0

Full release notes

15.8.1. Highlights

• This version underwent extensive testing and fixes regarding Large Messages, with a few JIRAs dedicated to this topic. Look on the release notes for more information.

15.8.2. Upgrading from 2.28.0

• Due to ARTEMIS-4151 the default access for MBeans not defined in the role-access or allowlist of management.xml is now read only. This is a precautionary measure to ensure no unanticipated MBean deployed with the broker poses a risk. However, this will also impact JVM-specific and platform MBeans as well (e.g. which allow manual garbage collection, "flight recording," etc.). Write access and general operational access to these MBeans will now have to be manually enabled in management.xml either by changing the default-access (not recommended) or specifically configuring a role-access for the particular MBean in question.

This applies to all MBean access including directly via JMX and via the Jolokia JMX-HTTP bridge.

• Due to ARTEMIS-4212 the broker will reject address definitions in broker.xml which don’t specify a routing type, e.g.:

  <address name="myAddress"/>

Such configurations will need to be changed to specify a routing-type, e.g.:

  <address name="myAddress">
    <anycast/>
  </address>

  Or

  <address name="myAddress">
    <multicast/>
  </address>
If an address without a routing type is configured the broker will throw an exception like this and fail to start:

```
java.lang.IllegalArgumentException: AMQ229247: Invalid address configuration for 'myAddress'. Address must support multicast and/or anycast.
    at org.apache.activemq.artemis.core.deployers.impl.FileConfigurationParser.parseAddresses(FileConfigurationParser.java:1580)
    at org.apache.activemq.artemis.core.deployers.impl.FileConfigurationParser.parseAddresses(FileConfigurationParser.java:1038)
    at org.apache.activemq.artemis.core.deployers.impl.FileConfigurationParser.parseMainConfig(FileConfigurationParser.java:804)
    at org.apache.activemq.artemis.core.config.impl.FileConfiguration.parse(FileConfiguration.java:56)
    at org.apache.activemq.artemis.core.config.FileDeploymentManager.readConfiguration(FileDeploymentManager.java:81)
    at org.apache.activemq.artemis.integration.FileBroker.createComponents(FileBroker.java:120)
    at org.apache.activemq.artemis.cli.commands.Run.execute(Run.java:119)
    at org.apache.activemq.artemis.cli.Artemis.internalExecute(Artemis.java:212)
    at org.apache.activemq.artemis.cli.Artemis.execute(Artemis.java:162)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(NativeMethod)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:566)
    at org.apache.activemq.artemis.boot.Artemis.execute(Artemis.java:144)
    at org.apache.activemq.artemis.boot.Artemis.main(Artemis.java:61)
```

- Due to ARTEMIS-3707 all use of `javax.transaction.TransactionManager` was removed from the JCA Resource Adapter. However, this rendered the `transactionTimeout` activation configuration property useless. Some existing users rely on this behavior so it has been restored and properly deprecated for future removal.

**15.9. 2.28.0**

Full release notes
15.9.1. Highlights

- Bug Fixes and improvements as usual
- **ARTEMIS-4136** Mirror sync replication
  - Mirror now has an option to set sync=true. Blocking operations from clients will wait a round trip on the mirror.
- **ARTEMIS-4065** Paging Counter Journal Records were removed
  - We don’t store page counters records on the journal any longer what should simplify operation and improve performance.

15.9.2. Upgrading from 2.27.0

- Due to **ARTEMIS-3871** the naming pattern used for MQTT *shared* subscription queues has changed. Previously the subscription queue was named according to the subscription name provided in the MQTT `SUBSCRIBE` packet. However, MQTT allows the same name to be used across multiple subscriptions whereas queues in the broker must be named uniquely. Now the subscription queue will be named according to the subscription name and topic name so that all subscription queue names will be unique. Before upgrading please ensure all MQTT shared subscriptions are empty. When the subscribers reconnect they will get a new subscription queue. If they are not empty you can move the messages to the new subscription queue administratively.

15.10. 2.27.1

Full release notes

15.10.1. Highlights

- Bug Fixes
- AMQP Large Message over Bridges were broken
- Rollback of massive transactions would take a long time to process
- Improvements to auto-create and auto-delete queues.

15.11. 2.27.0

Full release notes

15.11.1. Highlights

- 2.27.0 Introduced a new upgrade tool to help migrating your instance to a newer version.
- The client and broker now use SLF4J for their logging API.
- The broker distribution now uses Log4J 2 as its logging implementation.
15.11.2. Upgrading from 2.26.0

Client applications wanting logging will now need to supply an appropriate SLF4J-supporting logging implementation configured appropriately for their needs. See client application logging for more information plus an example around using Log4J 2.

The broker distribution now includes and configures Log4J 2 as its logging implementation, see logging for more details. If upgrading an existing broker instance rather than creating a new instance, some configuration etc updates will be necessary for the brokers existing instance /etc and /bin files.

You can use the new upgrade helper tool from the newly downloaded broker to refresh various configuration files and scripts for an existing broker instance. The broker.xml and data are left in place as-is.

You should back up your existing broker instance before running the command. The command can be executed by running "./artemis upgrade <path-to-your-instance>" from the new downloaded broker home.

Most existing customisations to the old configuration files and scripts will be lost in the process of refreshing the files. As such you should compare the old configuration files with the refreshed ones and then port any missing customisations you may have made as necessary. The upgrade command itself will copy the older files it changes to an old-config-bkp. folder within the instance directory.

Similarly, if you had customised the old logging.properties file you may need to prepare analogous changes for the new log4j2.properties file.

Note also that the configuration-file-refresh-period setting in broker.xml no longer covers logging configuration refresh. Log4J 2 has its own configuration reload handling, configured via the monitorInterval property within the Log4J configuration file itself. The default <instance>/etc/log4j2.properties file created has a 5 second monitorInterval value set to align with the prior default broker behaviour.

15.11.3. Manual update

Alternatively, rather than using the upgrade helper command as outlined above, you can instead perform the update manually, following the general upgrading procedure plus the additional steps below:

1. The new <instance>/etc/log4j2.properties file should be created with Log4J 2 configuration. The file used by the "artemis create" CLI command can be downloaded from: log4j2.properties
2. The old <instance>/etc/logging.properties JBoss Logging configuration file should be deleted.
3. Related startup script or profile cleanups are needed: a diff file demonstrating the changes needed since 2.26.0 is available here for *nix or here for Windows.
15.12. 2.26.0

Full release notes

15.12.1. Highlights

• Bug fixes and improvements

15.12.2. Upgrading from 2.25.0

1. Due to ARTEMIS-4006 the artemis-jms-client-all and artemis-jakarta-client-all clients were removed from the lib/client directory in the binary distribution. If you use these libraries they can be found at Maven Central (e.g. here). Please refer to the client class path documentation for more information.

2. We removed the REST interface from the code-base and documentation. If you still require the REST interface you can access the latest version which is still viable. You can still follow the steps from the previous documentation to build and deploy the interface. However, you should stop using it as it will not be maintained any more.

3. Due to ARTEMIS-3980 the web content was removed from the binary distribution. We now redirect web requests with the root target to the administration console. To enable this new redirect behavior on current instances you have to update bootstrap.xml. Change:

```xml
<web path="web">
```

to:

```xml
<web path="web" rootRedirectLocation="console">
```

If you used to customize the index page or to add custom content in the web folder please refer to the web-server documentation for more information on disabling the redirect and enabling the web content.

15.13. 2.25.0

Full release notes

15.13.1. Highlights

• Improvement on Paging Flow Control
• Many other bug fixes and improvements

15.14. 2.24.0

Full release notes
15.14.1. Highlights

- Streamlined page caches and files are just read into queues without the need of soft caches.

15.14.2. Upgrading from 2.23.0

1. Due to ARTEMIS-3851 the queue created for an MQTT 3.x subscriber using `CleanSession=1` is now non-durable rather than durable. This may impact security-settings for MQTT clients which previously only had `createDurableQueue` for their role. They will now need `createNonDurableQueue` as well. Again, this only has potential impact for MQTT 3.x clients using `CleanSession=1`.

2. Due to ARTEMIS-3892 the username assigned to queues will be based on the validated user rather than just the username submitted by the client application. This will impact use-cases like the following:
   a. When `login.config` is configured with the `GuestLoginModule` which causes some users to be assigned a specific username and role during the authentication process.
   b. When `login.config` is configured with the `CertificateLoginModule` which causes users to be assigned a username and role corresponding to the subject DN from their SSL certificate. In these kinds of situations the broker will use this assigned (i.e. validated) username for any queues created with the connection. In the past the queue's username would have been left blank.

15.15. 2.23.1

Full release notes

15.15.1. Highlights

- ARTEMIS-3856 - Failed to change channel state to ReadyForWriting : java.util.ConcurrentModificationException

15.16. 2.23.0

Full release notes.

15.16.1. Highlights

- management operations for the embedded web server.
- JakartaEE 10 Support
- BugFix: High cpu usage on ReadWrite locks

15.17. 2.22.0

Full release notes.
15.17.1. Highlights

- The default producer-window-size on cluster-connection was changed to 1MB to mitigate potential OutOfMemoryErrors in environments with high latency networking.

15.18. 2.21.0

Full release notes.

15.18.1. Highlights

- MQTT 5 is now supported.
- A new set of performance tools are now available to evaluate throughput and Response Under Load performance of Artemis
- Diverts now support multiple addresses
- Runtime configuration reloading now supports bridges.
- Paging can now be configured by message count.

15.18.2. Upgrading from 2.20.0

1. Due to XML schema changes to correct an inaccurate domain name 2 files will need to be updated:
   a. etc/bootstrap.xml
   b. etc/management.xml

   In both files change the XML namespace from activemq.org to activemq.apache.org, e.g. in bootstrap.xml use:
   
   ```xml
   <broker xmlns="http://activemq.apache.org/schema">
   ```

   And in management.xml use:
   
   ```xml
   <management-context xmlns="http://activemq.apache.org/schema">
   ```

2. If you're using JDBC persistence then due to the changes in ARTEMIS-3679 you'll need to update your database. The column HOLDER_EXPIRATION_TIME on the NODE_MANAGER_STORE changed from a TIMESTAMP to a BIGINT (or NUMBER(19) on Oracle). You will have to stop any broker that is accessing that table and either drop it or execute the proper ALTER TABLE statement for your database. If you drop the table then it will be automatically recreated when broker restarts and repopulated with a new, auto-generated node ID.

3. If you're using JGroups then due to the changes in ARTEMIS-2413 where JGroups was updated from 3.x to 5.x you will need to update your JGroups configuration. Many of the protocols have changed, and there's no automated tool to bring legacy configurations up to date so please refer to the JGroups documentation for more details on the new configuration. You can find example
configurations in the JGroups repository (e.g. tcp.xml and udp.xml).

15.19. 2.20.0

Full release notes.

15.19.1. Highlights

- Java 11 is now required.

15.20. 2.19.0

Full release notes.

15.20.1. Highlights

- New ability to replay retained journal records via the management API.
- New environment/system property to set the "key" for masked passwords when using the default codec.
- Ability to disable message-load-balancing and still allow redistribution via the new OFF_WITH_REDISTRIBUTION type.
- MQTT session state can now be cleaned up automatically to avoid excessive accumulation in situations where client's don't clean up their own sessions.
- Distribute full Jakarta Messaging 3.0 client in the lib/client directory along with a new example of how to use it in examples/features/standard/queue-jakarta.

15.21. 2.18.0

Full release notes.

15.21.1. Highlights

- Dual Mirror support improving capabilities on AMQP Mirror for Disaster Recovery
- Journal Retention
- Replication integrated with ZooKeeper
- Connection Routers
- Concurrency configuration for core bridges.
- XPath filter expressions (for parity with ActiveMQ "Classic").

15.21.2. Upgrading from 2.17.0

1. Due to ARTEMIS-3367 the default setting for verifyHost on core connectors has been changed from false to true. This means that core clients will now expect the CN or Subject Alternative Name values of the broker's SSL certificate to match the hostname in the client's URL.
This impacts all core-based clients including core JMS clients and core connections between cluster nodes. Although this is a "breaking" change, not performing hostname verification is a security risk (e.g. due to man-in-the-middle attacks). Enabling it by default aligns core client behavior with industry standards. To deal with this you can do one of the following:

- Update your SSL certificates to use a hostname which matches the hostname in the client's URL. This is the recommended option with regard to security.
- Update any connector using `sslEnabled=true` to also use `verifyHost=false`. Using this option means that you won't get the extra security of hostname verification, but no certificates will need to change. This essentially restores the previous default behavior.

For additional details about please refer to section 3.1 of RFC 2818 "HTTP over TLS".

2. Due to ARTEMIS-3117 SSL keystore and truststores are no longer reloaded automatically. Previously an instance of `javax.net.ssl.SSLContext` was created for *every* connection. This would implicitly pick up any changes to the keystore and truststore for any new connection. However, this was grossly inefficient and therefore didn't scale well with lots of connections. The behavior was changed so that just one `javax.net.ssl.SSLContext` is created for each acceptor. However, one can still reload keystores & truststores from disk without restarting the broker. Simply use the `reload` management operation on the acceptor. This is available via JMX, the web console, Jolokia, etc.

Here's an example `curl` command you can use with Jolokia to invoke the `artemis` acceptor's `reload` operation:

```bash
curl --user admin:admin --header "Content-Type: application/json" --request POST --data '{"type":"exec", "mbean":"org.apache.activemq.artemis:broker="0.0.0.0",component=acceptors,name="artemis"", "operation":"reload"}' http://localhost:8161/console/jolokia/exec
```

Of course you'll want to adjust the username & password as well as the broker and acceptor names for your environment.

3. The "rate" metric for queues was removed from the web console via ARTEMIS-3397. This was a follow-up from ARTEMIS-2909 in 2.16.0 (referenced in the upgrade instructions below). The "rate" metric mistakenly left visible on the web console after it was removed from the management API.

4. Due to ARTEMIS-3141, ARTEMIS-3128, & ARTEMIS-3175 the data returned for any "list" or "browse" management method which return message data, including those exposed via the web console, will have their return data truncated by default. This is done to avoid adverse conditions with large volumes of message data which could potentially negatively impact broker stability. The `management-message-attribute-size-limit` address-setting controls this behavior. If you wish to restore the previous (and potentially dangerous behavior) then you can specify `-1` for this. It is 256 by default.
15.22. 2.17.0

Full release notes.

15.22.1. Highlights

- **Message-level authorization** similar to ActiveMQ "Classic".
- A count of addresses and queues is now available from the management API.
- You can now reload the broker's configuration from disk via the management API rather than waiting for the periodic disk scan to pick it up
- Performance improvements on libaio journal.
- New command-line option to transfer messages.
- Performance improvements for the wildcard address manager.
- JDBC datasource property values can now be masked.
- Lots of usability improvements to the Hawtio 2 based web console introduced in 2.16.0
- New management method to create a core bridge using JSON-based configuration input.
- Jakarta Messaging 2.0 & 3.0 artifacts for Jakarta EE 8 & 9 respectively.

15.23. 2.16.0

Full release notes.

15.23.1. Highlights

- Configurable namespace for temporary queues
- AMQP Server Connectivity
- "Basic" SecurityManager implementation that supports replication
- Consumer window size support for individual STOMP clients
- Improved JDBC connection management
- New web console based on Hawtio 2
- Performance optimizations (i.e. caching) for authentication and authorization
- Support for admin objects in the JCA resource adapter to facilitate deployment into 3rd-party Java EE application servers
- Ability to prevent an acceptor from automatically starting

15.23.2. Upgrading from 2.15.0

1. Due to ARTEMIS-2893 the fundamental way user management was implemented had to change to avoid data integrity issues related to concurrent modification. From a user's perspective two main things changed:

   a. User management is no longer possible using the **artemis user** commands when the broker
is **offline**. Of course users are still free to modify the properties files directly in this situation.

b. The parameters of the *artemis user* commands changed. Instead of using something like this:

```bash
./artemis user add --user guest --password guest --role admin
```

Use this instead:

```bash
./artemis user add --user-command-user guest --user-command-password guest --role admin
```

In short, use *user-command-user* in lieu of *user* and *user-command-password* in lieu of *password*. Both *user* and *password* parameters now apply to the connection used to send the command to the broker.

For additional details see ARTEMIS-2893 and ARTEMIS-3010

2. Due to ARTEMIS-2909 the "rate" metric was removed from the management API for queues. In short, the `org.apache.activemq.artemis.core.server.Queue#getRate` method is for slow-consumer detection and is designed for *internal* use only.

Furthermore, it's too opaque to be trusted by a remote user as it only returns the number of message added to the queue since *the last time it was called*. The problem here is that the user calling it doesn't know when it was invoked last. Therefore, they could be getting the rate of messages added for the last 5 minutes or the last 5 milliseconds. This can lead to inconsistent and misleading results.

There are three main ways for users to track rates of message production and consumption (in recommended order):

a. Use a **metrics** plugin. This is the most feature-rich and flexible way to track broker metrics, although it requires tools (e.g. Prometheus) to store the metrics and display them (e.g. Grafana).

b. Invoke the `getMessageCount()` and `getMessagesAdded()` management methods and store the returned values along with the time they were retrieved. A time-series database is a great tool for this job. This is exactly what tools like Prometheus do. That data can then be used to create informative graphs, etc. using tools like Grafana. Of course, one can skip all the tools and just do some simple math to calculate rates based on the last time the counts were retrieved.

c. Use the broker’s **message counters**. Message counters are the broker’s simple way of providing historical information about the queue. They provide similar results to the previous solutions, but with less flexibility since they only track data while the broker is up and there’s not really any good options for graphing.
15.24. 2.15.0

Full release notes.

15.24.1. Highlights

• Ability to use FQQN syntax for both security-settings and JNDI lookup
• Support pausing dispatch during group rebalance (to avoid potential out-of-order consumption)
• Socks5h support

15.25. 2.14.0

Full release notes.

15.25.1. Highlights

• Management methods to update diverts
• Ability to "disable" a queue so that messages are not routed to it
• Support JVM GC & thread metrics
• Support for resetting queue properties by unsetting them in broker.xml
• Undeploy diverts by removing them from broker.xml
• Add addressMemoryUsagePercentage and addressSize as metrics

15.25.2. Upgrading from 2.13.0

This is likely a rare situation, but it's worth mentioning here anyway. Prior to 2.14.0 if you configured a parameter on a queue in broker.xml (e.g. max-consumers) and then later removed that setting the configured value you set would remain. This has changed in 2.14.0 via ARTEMIS-2797. Any value that is not explicitly set in broker.xml will be set back to either the static default or the dynamic default configured in the address-settings (e.g. via default-max-consumers in this example). Therefore, ensure any existing queues have all the needed parameters set in broker.xml values before upgrading.

15.26. 2.13.0

Full release notes.

15.26.1. Highlights

• Management methods for an address' duplicate ID cache to check the cache's size and clear it
• Support for min/max expiry-delay
• Per-acceptor security domains
• Command-line check tool for checking the health of a broker
• Support disabling metrics per address via the `enable-metrics` address setting
• Improvements to the `audit logging`
• Speed optimizations for the `HierarchicalObjectRepository`, an internal object used to store address and security settings

15.26.2. Upgrading from 2.12.0

Version 2.13.0 added new `audit logging` which is logged at `INFO` level and can be very verbose. The `logging.properties` shipped with this new version is set up to filter this out by default. If your `logging.properties` isn’t updated appropriately this audit logging will likely appear in your console and `artemis.log` file assuming you’re using a logging configuration close to the default. Add this to your `logging.properties`:

```plaintext
# to enable audit change the level to INFO
logger.org.apache.activemq.audit.base.level=ERROR
logger.org.apache.activemq.audit.base.handlers=AUDIT_FILE
logger.org.apache.activemq.audit.base.useParentHandlers=false

logger.org.apache.activemq.audit.resource.level=ERROR
logger.org.apache.activemq.audit.resource.handlers=AUDIT_FILE
logger.org.apache.activemq.audit.resource.useParentHandlers=false

logger.org.apache.activemq.audit.message.level=ERROR
logger.org.apache.activemq.audit.message.handlers=AUDIT_FILE
logger.org.apache.activemq.audit.message.useParentHandlers=false

... 

#Audit logger
handler.AUDIT_FILE=org.jboss.logmanager.handlers.PeriodicRotatingFileHandler
handler.AUDIT_FILE.level=INFO
handler.AUDIT_FILE.properties=suffix,append,autoFlush,fileName
handler.AUDIT_FILE.suffix=.yyyy-MM-dd
handler.AUDIT_FILE.append=true
handler.AUDIT_FILE.autoFlush=true
handler.AUDIT_FILE.fileName=${artemis.instance}/log/audit.log
handler.AUDIT_FILE.formatter=AUDIT_PATTERN

formatter.AUDIT_PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.AUDIT_PATTERN.properties=pattern
formatter.AUDIT_PATTERN.pattern=%d [AUDIT](%t) %s%E%n
```

15.27. 2.12.0

Full release notes.
15.27.1. Highlights

- Support for **SOCKS proxy**
- Real **large message** support for AMQP
- **Automatic creation of dead-letter resources** akin to ActiveMQ 5’s individual dead-letter strategy
- **Automatic creation of expiry resources**
- Improved API for queue creation
- Allow users to override JAVA_ARGS via environment variable
- Reduce heap usage during journal loading during broker start-up
- Allow **server** header in STOMP **CONNECTED** frame to be disabled
- Support disk store used percentage as an exportable metric (e.g. to be monitored by tools like Prometheus, etc.)
- Ability to configure a "**customizer**" for the embedded web server
- Improved logging for errors when starting an **acceptor** to more easily identify the **acceptor** which has the problem.
- The CLI will now read the **broker.xml** to find the default **connector** URL for commands which require it (e.g. **consumer**, **producer**, etc.)

15.28. 2.11.0

**Full release notes.**

15.28.1. Highlights

- Support **retroactive addresses**.
- Support downstream federated **queues** and **addresses**.
- Make security manager **configurable via XML**.
- Support pluggable SSL **TrustManagerFactory**.
- Add plugin support for federated queues/addresses.
- Support **com.sun.jndi.ldap.read.timeout** in **LDAPLoginModule**.

15.29. 2.10.0

**Full release notes.**

This was mainly a bug-fix release with a notable dependency change impacting version upgrade.

15.29.1. Upgrading from 2.9.0

Due to the WildFly dependency upgrade the broker start scripts/configuration need to be adjusted after upgrading.
On *nix
Locate this statement in `bin/artemis`:

```bash
WILDFLY_COMMON="$ARTEMIS_HOME/lib/wildfly-common-1.5.1.Final.jar"
```

This needs to be replaced with this:

```bash
WILDFLY_COMMON="$ARTEMIS_HOME/lib/wildfly-common-1.5.2.Final.jar"
```

On Windows
Locate this part of `JAVA_ARGS` in `etc/artemis.profile.cmd` respectively `bin/artemis-service.xml`:

```bash
%ARTEMIS_HOME%\lib\wildfly-common-1.5.1.Final.jar
```

This needs to be replaced with this:

```bash
%ARTEMIS_HOME%\lib\wildfly-common-1.5.2.Final.jar
```

### 15.30. 2.9.0

**Full release notes.**

This was a light release. It included a handful of bug fixes, a few improvements, and one major new feature.

#### 15.30.1. Highlights

- Support exporting metrics.

### 15.31. 2.8.1

**Full release notes.**

This was mainly a bug-fix release with a notable dependency change impacting version upgrade.

#### 15.31.1. Upgrading from 2.8.0

Due to the dependency upgrade made on ARTEMIS-2319 the broker start scripts need to be adjusted after upgrading.

On *nix
Locate this if statement in `bin/artemis`:
if [ -z "$LOG_MANAGER" ] ; then
  # this is the one found when the server was created
  LOG_MANAGER="$ARTEMIS_HOME/lib/jboss-logmanager-2.0.3.Final.jar"
fi

This needs to be replaced with this block:

if [ -z "$LOG_MANAGER" ] ; then
  # this is the one found when the server was created
  LOG_MANAGER="$ARTEMIS_HOME/lib/jboss-logmanager-2.1.10.Final.jar"
fi

WILDFLY_COMMON=`ls $ARTEMIS_HOME/lib/wildfly-common*jar 2>/dev/null`
if [ -z "$WILDFLY_COMMON" ] ; then
  # this is the one found when the server was created
  WILDFLY_COMMON="$ARTEMIS_HOME/lib/wildfly-common-1.5.1.Final.jar"
fi

Notice that the jboss-logmanager version has changed and there is also a new wildfly-common library.

Not much further down there is this line:

-XXbootclasspath/a:"$LOG_MANAGER" \\

This line should be changed to be:

-XXbootclasspath/a:"$LOG_MANAGER:$WILDFLY_COMMON" \\

On Windows

Locate this part of JAVA_ARGS in etc/artemis.profile.cmd respectively bin/artemis-service.xml:

-XXbootclasspath/a:%ARTEMIS_HOME%\lib\jboss-logmanager-2.1.10.Final.jar

This needs to be replaced with this:

-XXbootclasspath/a:%ARTEMIS_HOME%\lib\jboss-logmanager
-2.1.10.Final.jar;%ARTEMIS_HOME%\lib\wildfly-common-1.5.1.Final.jar

15.32. 2.8.0

Full release notes.
15.32.1. Highlights

- Support ActiveMQ feature JMSXGroupFirstForConsumer.
- Clarify handshake timeout error with remote address.
- Support duplicate detection for AMQP messages the same as core.

15.33. 2.7.0
Full release notes.

15.33.1. Highlights

- Support advanced destination options like consumersBeforeDispatchStarts and timeBeforeDispatchStarts from 5.x.
- Add support for delays before deleting addresses and queues via auto-delete-queues-delay and auto-delete-addresses-delay Address Settings.
- Support logging HTTP access.
- Add a CLI command to purge a queue.
- Support user and role manipulation for PropertiesLoginModule via management interfaces.
- Docker images.
- Audit logging.
- Implementing consumer priority.
- Support FQCN for producers.
- Track routed and unrouted messages sent to an address.
- Support connection pooling in LDAPLoginModule.
- Support configuring a default consumer window size via default-consumer-window-size Address Setting.
- Support masking key-store-password and trust-store-password in management.xml.
- Support JMSXGroupSeq -1 to close/reset message groups from 5.x.
- Allow configuration of RMI registry port.
- Support routing-type configuration on core bridge.
- Move artemis-native as its own project, as activemq-artemis-native.
- Support federated queues and addresses.

15.34. 2.6.4
Full release notes.

This was mainly a bug-fix release with a few improvements a couple notable new features:
15.34.1. Highlights

- Added the ability to set the text message content on the producer CLI command.
- Support reload logging configuration at runtime.

15.35. 2.6.3

Full release notes.

This was mainly a bug-fix release with a few improvements but no substantial new features.

15.36. 2.6.2

Full release notes.

This was a bug-fix release with no substantial new features or improvements.

15.37. 2.6.1

Full release notes.

This was a bug-fix release with no substantial new features or improvements.

15.38. 2.6.0

Full release notes.

15.38.1. Highlights

- Support regular expressions for matching client certificates.
- Support SASL_EXTERNAL for AMQP clients.
- New examples showing virtual topic mapping and exclusive queue features.

15.39. 2.5.0

Full release notes.

15.39.1. Highlights

- Exclusive consumers.
- Equivalent ActiveMQ "Classic" Virtual Topic naming abilities.
- SSL Certificate revocation list.
- Last-value queue support for OpenWire.
- Support masked passwords in bootstrap.xm and login.config
• Configurable broker plugin implementation for logging various broker events (i.e. LoggingActiveMQServerPlugin).
• Option to use OpenSSL provider for Netty via the sslProvider URL parameter.
• Enable splitting of broker.xml into multiple files.
• Enhanced message count and size metrics for queues.

15.39.2. Upgrading from 2.4.0

1. Due to changes from ARTEMIS-1644 any acceptor that needs to be compatible with HornetQ and/or Artemis 1.x clients needs to have ancastPrefix=jms.queue.;multicastPrefix=jms.topic. in the acceptor url. This prefix used to be configured automatically behind the scenes when the broker detected these old types of clients, but that broke certain use-cases with no possible work-around. See ARTEMIS-1644 for more details.

15.40. 2.4.0

Full release notes.

15.40.1. Highlights

• JMX configuration via XML rather than having to use system properties via command line or start script.
• Configuration of max frame payload length for STOMP web-socket.
• Ability to configure HA using JDBC persistence.
• Implement role-based access control for management objects.

15.40.2. Upgrading from 2.3.0

1. Create <ARTEMIS_INSTANCE>/etc/management.xml. At the very least, the file must contain this:

```xml
<management-context xmlns="http://activemq.apache.org/schema"/>
```

This configures role based authorisation for JMX. Read more in the Management documentation.

2. If configured, remove the Jolokia war file from the web element in <ARTEMIS_INSTANCE>/etc/bootstrap.xml:

```xml
<app url="jolokia" war="jolokia.war"/>
```

This is no longer required as the Jolokia REST interface is now integrated into the console web application.

If the following is absent and you desire to deploy the web console then add:
the Jolokia REST interface URL will now be at http://<host>:<port>/console/jolokia

15.41. 2.3.0

Full release notes.

15.41.1. Highlights

- Web admin console!
- Critical Analysis and deadlock detection on broker
- Support Netty native kqueue on Mac.
- Last-value queue for AMQP

15.41.2. Upgrading from 2.2.0

1. If you desire to deploy the web console then add the following to the web element in <ARTEMIS_INSTANCE>/etc/bootstrap.xml:

   `<app url="console" war="console.war"/>

15.42. 2.2.0

Full release notes.

15.42.1. Highlights

- Scheduled messages with the STOMP protocol.
- Support for JNDIReferenceFactory and JNDIStorable.
- Ability to delete queues and addresses when broker.xml changes.
- Client authentication via Kerberos TLS Cipher Suites (RFC 2712).

2.1.0

Full release notes.

15.42.2. Highlights

- Broker plugin support.
- Support Netty native epoll on Linux.
• Ability to configure arbitrary security role mappings.
• AMQP performance improvements.

15.43. 2.0.0
Full release notes.

15.43.1. Highlights
• Huge update involving a significant refactoring of the addressing model yielding the following benefits:
  ◦ Simpler and more flexible XML configuration.
  ◦ Support for additional messaging use-cases.
  ◦ Eliminates confusing JMS-specific queue naming conventions (i.e. "jms.queue." & "jms.topic." prefixes).
• Pure encoding of messages so protocols like AMQP don't need to convert messages to "core" format unless absolutely necessary.
• "MAPPED" journal type for increased performance in certain use-cases.

15.44. 1.5.6
Full release notes.

15.44.1. Highlights
• Bug fixes.

15.45. 1.5.5
Full release notes.

15.45.1. Highlights
• Bug fixes.

15.46. 1.5.4
Full release notes.

15.46.1. Highlights
• Support Oracle12C for JDBC persistence.
• Bug fixes.
15.47. 1.5.3
Full release notes.

15.47.1. Highlights
• Support "byte notation" (e.g. "K", "KB", "Gb", etc.) in broker XML configuration.
• CLI command to recalculate disk sync times.
• Bug fixes.

15.48. 1.5.2
Full release notes.

15.48.1. Highlights
• Support for paging using JDBC.
• Bug fixes.

15.49. 1.5.1
Full release notes.

15.49.1. Highlights
• Support outgoing connections for AMQP.
• Bug fixes.

15.50. 1.5.0
Full release notes.

15.50.1. Highlights
• AMQP performance improvements.
• JUnit rule implementation so messaging resources like brokers can be easily configured in tests.
• Basic CDI integration.
• Store user’s password in hash form by default.

15.51. 1.4.0
Full release notes.
15.51.1. Highlights

- "Global" limit for disk usage.
- Detect and reload certain XML configuration changes at runtime.
- MQTT interceptors.
- Support adding/deleting queues via CLI.
- New "browse" security permission for clients who only wish to look at messages.
- Option to populate JMSXUserID.
- "Dual authentication" support to authenticate SSL-based and non-SSL-based clients differently.

15.52. 1.3.0

Full release notes.

15.52.1. Highlights

- Better support of OpenWire features (e.g. reconnect, producer flow-control, optimized acknowledgements)
- SSL keystore reload at runtime.
- Initial support for JDBC persistence.
- Support scheduled messages on last-value queue.

15.53. 1.2.0

Full release notes.

15.53.1. Highlights

- Improvements around performance
- OSGi support.
- Support functionality equivalent to all 5.x JAAS login modules including:
  - Properties file
  - LDAP
  - SSL certificate
  - "Guest"

15.54. 1.1.0

Full release notes.
15.54.1. Highlights

- MQTT support.
- The examples now use the CLI programmatically to create, start, stop, etc. servers reflecting real cases used in production.
- CLI improvements. There are new tools to compact the journal and additional improvements to the user experience.
- Configurable resource limits.
- Ability to disable server-side message load-balancing.

15.55. 1.0.0

Full release notes.

15.55.1. Highlights

- First release of the donated code-base as ActiveMQ Artemis!
- Lots of features for parity with ActiveMQ “Classic” including:
  - OpenWire support
  - AMQP 1.0 support
  - URL based connections
  - Auto-create addresses/queues
  - Jolokia integration
Chapter 16. Upgrading the Broker

Apache ActiveMQ "Classic" (and previous versions) is runnable out of the box by executing the command: 

```
./bin/activemq run
```

The ActiveMQ Artemis broker follows a different paradigm where the project distribution serves as the broker "home" and one or more broker "instances" are created which reference the "home" for resources (e.g. jar files) which can be safely shared between broker instances. Therefore, an instance of the broker must be created before it can be run. This may seems like an overhead at first glance, but it becomes very practical when updating to a new Artemis version for example.

To create an Artemis broker instance navigate into the Artemis home folder and run: 

```
./bin/artemis create /path/to/myBrokerInstance
```

Because of this separation it's very easy to upgrade Artemis in most cases.

It's recommended to choose a folder different from where Apache Artemis was downloaded. This separation allows you run multiple broker instances with the same Artemis "home" for example. It also simplifies updating to newer versions of Artemis.

16.1. General Upgrade Procedure

Upgrading may require some specific steps noted in the versions, but the general process is as follows:

1. Navigate to the etc folder of the broker instance that's being upgraded
2. Open artemis.profile (artemis.profile.cmd on Windows). It contains a property which is relevant for the upgrade:

```
ARTEMIS_HOME='/path/to/apache-artemis-version'
```

If you run Artemis as a service on windows you have to do the following additional steps:

1. Navigate to the bin folder of the broker instance that's being upgraded
2. Open artemis-service.xml. It contains a property which is relevant for the upgrade:

```
<env name="ARTEMIS_HOME" value="/path/to/apache-artemis-version"/>
```

The ARTEMIS_HOME property is used to link the instance with the home. In most cases the instance can be upgraded to a newer version simply by changing the value of this property to the location of the new broker home. Please refer to the aforementioned versions document for additional upgrade steps (if required).

It is also possible to do many of these update steps automatically as can be seen in the next section.
16.2. Upgrading tool

An upgrade helper tool from the new broker download can be used to refresh various configuration files and scripts from an existing broker instance from a prior version, and thus automate much of work to upgrade the instance to use the new version.

⚠ You should back up your existing broker instance before running the command.

```bash
cd $NEW_ARTEMIS_DOWNLOAD/bin/
./artemis upgrade PATH_TO_UPGRADING_INSTANCE
```

The broker instance `bin/artemis` script and `etc/artemis.profile` (and `artemis.cmd` and `artemis.cmd.profile` on Windows) will be updated to the new versions, setting its `ARTEMIS_HOME` to refer to the new broker version home path. The tool will also create the new `<instance>/etc/log4j2.properties` configuration file if needed (e.g. if you are migrating from a version prior to 2.27.0), and remove the old `<instance>/etc/logging.properties` file if present.

The `broker.xml` file and data are retained as-is.

⚠ Most existing customisations to the old configuration files and scripts will be lost in the process of refreshing the files. As such you should compare the old configuration files with the refreshed ones and then port any missing customisations you may have made as necessary. The upgrade command itself will copy the older files it changes to an `old-config-bkp` folder within the instance dir.

Similarly, if you had customised the old `logging.properties` file you may need to prepare analogous changes for the new `log4j2.properties` file.
Chapter 17. Docker

One of the simplest ways to get started with ActiveMQ Artemis is by using one of our Docker images.

17.1. Official Images

Official Docker images are available on dockerhub. Images are pushed along with all the other distribution artifacts for every release. The fastest, simplest way to get started is with this command which will create and start a detached container named mycontainer, expose the main messaging port (i.e. 61616) and HTTP port (i.e. 8161), and remove it when it terminates:

```
```

Once the broker starts you can open the web management console at http://localhost:8161 and log in with the default username & password artemis.

You can also use the shell command to interact with the running broker using the default username & password artemis, e.g.:

```
$ docker exec -it mycontainer /var/lib/artemis-instance/bin/artemis shell --user artemis --password artemis
```

Using the shell command you can execute basic tasks like creating & deleting addresses and queues, sending and browsing messages, viewing queue statistics, etc. See the Command Line Interface chapter for more details.

You can view the container’s logs using:

```
$ docker logs -f mycontainer
```

Stop the container using:

```
$ docker stop mycontainer
```

The official Docker images are built using these scripts which you can also use to build your own images. Read on for more details.

17.2. Build your own Image

In order to build an image you need an ActiveMQ Artemis binary distribution. This can be sourced locally (in which case you need to build the project first) or remotely based on an official Apache release.
17.2.1. Using a Local Release

If you want to use a local binary distribution then build it from the root of the ActiveMQ source tree, e.g.:

\[
\text{mvn -Prelease package -DskipTests}
\]

Following the build the distribution files will be in your local distribution directory. Here \(<version>\) is the version of the distribution that you built.

\[
\text{artemis-distribution/target/apache-artemis-<version>-bin/apache-artemis-<version>}
\]

Then switch to the \(\text{artemis-docker}\) directory and use the \(\text{prepare-docker.sh}\) script with the proper parameters to copy the Docker files into your local binary distribution, e.g.:

\[
\text{cd \(\text{artemis-docker}\)}
\]
\[
\text{./prepare-docker.sh --from-local-dist --local-dist-path ..} / \text{artemis-distribution/target/apache-artemis-<version>-bin/apache-artemis-<version>}
\]

This will copy all the files necessary to build any of the pre-configured Docker images and provide you with additional instructions. Follow these instructions to finish building the image you want based on one of the provided Docker files or even one of your own.

17.2.2. Using an Official Apache Release

If you would rather use an official Apache release in your image rather than a local release then run the following command from the \(\text{artemis-docker}\) directory where \(<version>\) is the release version you wish to use (e.g. 2.30.0):

\[
\text{./prepare-docker.sh --from-release --artemis-version <version>}
\]

This will copy all the files necessary to build any of the pre-configured Docker images and provide you with additional instructions. Follow these instructions to finish building the image you want based on one of the provided Docker files or even one of your own.

17.2.3. Customizing the Image

Environment Variables

Environment variables determine the options configured for the \(\text{artemis create}\) command when running \(\text{docker build}\). The available options are:

\text{ARTEMIS_USER}

The administrator username. The default is \text{artemis}. 

...
**ARTEMIS_PASSWORD**

The administrator password. The default is *artemis*.

**ANONYMOUS_LOGIN**

Set to *true* to allow anonymous logins. The default is *false*.

**EXTRA_ARGS**

Additional arguments sent to the *artemis create* command. The default is *--http-host 0.0.0.0 --relax-jolokia*. Setting this value will override the default. See the documentation on *artemis create* for available options.

The combination of the above environment variables results in the *docker-run.sh* script calling the following command to create the broker instance the first time the Docker container runs:

```
${ARTEMIS_HOME}/bin/artemis create --user ${ARTEMIS_USER} --password ${ARTEMIS_PASSWORD} --silent $LOGIN_OPTION $EXTRA_ARGS
```

Note: *LOGIN_OPTION* is either *--allow-anonymous* or *--require-login* depending on the value of *ANONYMOUS_LOGIN*.

These variables can be set in the relevant Dockerfile or, for example, on the command-line, e.g.:

```
$ docker run -e ARTEMIS_USER=myUser -e ARTEMIS_PASSWORD=myPass --name mycontainer -it -p 61616:61616 -p 8161:8161 apache/activemq-artemis:latest-alpine
```

**Mapping point**

The image will use the directory */var/lib/artemis-instance* to hold the configuration and the data of the running broker. You can map this to a folder on the host for when you want the configuration and data persisted outside of a container, e.g.:

```
docker run -it -p 61616:61616 -p 8161:8161 -v <broker folder on host>:/var/lib/artemis-instance apache/activemq-artemis:latest-alpine
```

In this case the value `<broker folder on host>` is a directory where the broker instance is supposed to be saved and reused on each run.

**Overriding files in etc folder**

You can use customized configuration for the ActiveMQ Artemis instance by replacing the files residing in the *etc* folder with the custom ones, e.g. *broker.xml* or *artemis.profile*. Put the replacement files inside a folder and map it as a volume to:

```
/var/lib/artemis-instance/etc-override
```
The contents of `etc-override` folder will be copied over to `etc` folder after the instance creation so that the broker will always start with user-supplied configuration.

It you are mapping the whole `var/lib/artemis-instance` to an outside folder for persistence then you can place an `etc-override` folder inside the mapped one. Its contents will again be copied over `etc` folder after creating the instance.
Chapter 18. Using the Server

This chapter will familiarise you with how to use the Apache ActiveMQ Artemis server.

We'll show where it is, how to start and stop it, and we'll describe the directory layout and what all the files are and what they do.

This document will refer to the full path of the directory where the ActiveMQ distribution has been extracted to as \( \texttt{${ARTEMIS_HOME}$} \).

18.1. Installation

You can get the latest release from the Download page.

The following highlights some important folders on the distribution:

```
|   ___ bin
|   |   ___ lib
|   |   |   ___ schema
|   |   ___ web
```

**bin**
- binaries and scripts needed to run ActiveMQ Artemis.

**lib**
- jars and libraries needed to run ActiveMQ Artemis

**schema**
- XML Schemas used to validate ActiveMQ Artemis configuration files

**web**
- The folder where the web context is loaded when the broker runs.

18.2. Creating a Broker Instance

A broker *instance* is the directory containing all the configuration and runtime data, such as logs and message journal, associated with a broker process. It is recommended that you do *not* create the instance directory under \( \texttt{${ARTEMIS_HOME}$} \). This separation is encouraged so that you can more easily upgrade when the next version of ActiveMQ Artemis is released.

On Unix systems, it is a common convention to store this kind of runtime data under the \texttt{/var/lib} directory. For example, to create an instance at \texttt{/var/lib/mybroker}, run the following commands in your command line shell:
Before the broker is used, a broker instance must be created. This process requires the use of the Command Line Interface which is better explained in its own chapter.

In the following example a broker instance named `mybroker` will be created:

```bash
$ cd /var/lib
$ ${ARTEMIS_HOME}/bin/artemis create mybroker
```

A broker instance directory will contain the following sub directories:

**bin**

- holds execution scripts associated with this instance.

**data**

- holds the data files used for storing persistent messages

**etc**

- hold the instance configuration files

**lib**

- holds any custom runtime Java dependencies like transformers, plugins, interceptors, etc.

**log**

- holds rotating log files

**tmp**

- holds temporary files that are safe to delete between broker runs

At this point you may want to adjust the default configuration located in the `etc` directory.

### 18.2.1. Options

There are several options you can use when creating an instance. For a full list of options use the `help` command:

```bash
$ ./artemis help create
```
Create a new broker instance.

```bash
<directory>
```

The instance directory to hold the broker's configuration and data. Path must be writable.

```bash
--addresses=<addresses>
```
A comma separated list of addresses with the option to specify a routing type, e.g. --addresses myAddress1,myAddress2:anycast. Routing-type default: multicast.

```bash
--aio
```
Set the journal as asyncio.

```bash
--allow-anonymous
```
Allow connections from users with no security credentials. Opposite of --require-login. Default: input.

```bash
--autocreate
```
Allow automatic creation of addresses & queues. Default: true.

```bash
--autodelete
```
Allow automatic deletion of addresses & queues. Default: false.

```bash
--backup
```
Be a backup broker. Valid for shared store or replication.

```bash
--blocking
```
Block producers when address becomes full.
--cluster-password=<clusterPassword>
The password to use for clustering. Default: input.

--cluster-user=<clusterUser>
The user to use for clustering. Default: input.

--clustered
Enable clustering.

--data=<data>
Directory where ActiveMQ data are stored. Paths can be absolute or relative to artemis.instance directory. Default: data.

--default-port=<defaultPort>
The port number to use for the main 'artemis' acceptor. Default: 61616.

--disable-persistence
Disable message persistence to the journal

--encoding=<encoding>
The encoding that text files should use. Default: UTF-8.

--etc=<etc>
Directory where ActiveMQ configuration is located. Paths can be absolute or relative to artemis.instance directory. Default: etc.

--failover-on-shutdown
Whether broker shutdown will trigger failover for clients using the core protocol. Valid only for shared store. Default: false.

--force
Overwrite configuration at destination directory.

--global-max-messages=<globalMaxMessages>
Maximum number of messages that will be accepted in memory before using address full policy mode. Default: undefined.

--global-max-size=<globalMaxSize>
Maximum amount of memory which message data may consume. Default: half of the JVM's max memory.

--home=<home>
Directory where ActiveMQ Artemis is installed.

--host=<host>
Broker's host name. Default: 0.0.0.0 or input if clustered.

--http-host=<httpHost>
Embedded web server's host name. Default: localhost.

--http-port=<httpPort>
Embedded web server's port. Default: 8161.

--java-memory=<javaMemory>
Define the -Xmx memory parameter for the broker. Default: 2G.

--java-options=<javaOptions>
Extra Java options to be passed to the profile.

--jdbc
Store message data in JDBC instead of local files.

--jdbc-bindings-table-name=<jdbcBindings>
Name of the jdbc bindings table.

--jdbc-connection-url=<jdbcURL>
The URL used for the database connection.

--jdbc-driver-class-name=<jdbcClassName>
JDBC driver classname.

--jdbc-large-message-table-name=<jdbcLargeMessages>
Name of the large messages table.

--jdbc-lock-expiration=<jdbcLockExpiration>
Lock expiration (in milliseconds).
--jdbc-lock-renew-period=<jdbcLockRenewPeriod>
Lock Renew Period (in milliseconds).
--jdbc-message-table-name=<jdbcMessages>
Name of the jdbc messages table.
--jdbc-network-timeout=<jdbcNetworkTimeout>
Network timeout (in milliseconds).
--jdbc-node-manager-table-name=<jdbcNodeManager>
Name of the jdbc node manager table.
--jdbc-page-store-table-name=<jdbcPageStore>
Name of the page store messages table.
--journal-device-block-size=<journalDeviceBlockSize>
The block size of the journal's storage device.
Default: 4096.
--journal-retention=<retentionDays>
Configure journal retention in days. If > 0 then
enable journal-retention-directory from broker.
xml allowing replay options.
--journal-retention-max-bytes=<retentionMaxBytes>
Maximum number of bytes to keep in the retention
directory.
--linux, --cygwin  Force Linux or Cygwin script creation. Default:
based on your actual system.
--mapped  Set the journal as mapped.
--max-hops=<maxHops>  Number of hops on the cluster configuration.
--message-load-balancing=<messageLoadBalancing>
Message load balancing policy for cluster.
Default: ON_DEMAND. Valid values: ON_DEMAND,
STRICT, OFF, OFF_WITH_REDISTRIBUTION.
--name=<name>
The name of the broker. Default: same as host name.
--nio  Set the journal as nio.
--no-amqp-acceptor  Disable the AMQP specific acceptor.
--no-autocreate  Disable auto creation for addresses & queues.
--no-autotune  Disable auto tuning of the journal-buffer-timeout
in broker.xml.
--no-fsync  Disable usage of fdatasync (channel.force(false)
from Java NIO) on the journal.
--no-hornetq-acceptor  Disable the HornetQ specific acceptor.
--no-mqtt-acceptor  Disable the MQTT specific acceptor.
--no-stomp-acceptor  Disable the STOMP specific acceptor.
--no-web  Whether to omit the web-server definition from
bootstrap.xml.
--paging  Page messages to disk when address becomes full.
Opposite of --blocking. Default: true.
--password=<password>  The user's password. Default: input.
--ping=<ping>  A comma separated string to be passed on to the
broker config as network-check-list. The broker
will shutdown when all these addresses are
unreachable.
--port-offset=<portOffset>  How much to off-set the ports of every acceptor.
--queues=<queues>  A comma separated list of queues with the option
Some of these options may be mandatory in certain configurations and the system may ask you for additional input, e.g.:

```
$ ./artemis create /usr/server
Creating ActiveMQ Artemis instance at: /usr/server

--user: What is the default username?
admin

--password: is mandatory with this configuration:
```
What is the default password?

--allow-anonymous | --require-login:
Allow anonymous access?, valid values are Y, N, True, False

Y

Auto tuning journal ...
done! Your system can make 250 writes per millisecond, your journal-buffer-timeout
will be 4000

You can now start the broker by executing:

"/usr/server" run

Or you can run the broker in the background using:

"/usr/server" start

18.3. Starting and Stopping a Broker Instance

Assuming you created the broker instance under /var/lib/mybroker all you need to do start running
the broker instance is execute:

/var/lib/mybroker/bin/artemis run

To stop the Apache ActiveMQ Artemis instance you will use the same artemis script, but with the
stop argument. Example:

/var/lib/mybroker/bin/artemis stop

Please note that Apache ActiveMQ Artemis requires a Java 11 or later.

By default the etc/bootstrap.xml configuration is used. The configuration can be changed e.g. by
running ./artemis run -- xml:path/to/bootstrap.xml or another config of your choosing.

Environment variables are used to provide ease of changing ports, hosts and data directories used
and can be found in etc/artemis.profile on linux and etc\artemis.profile.cmd on Windows.

18.4. Configuration Files

These are the files you're likely to find in the etc directory of a default broker instance with a short
explanation of what they configure. Scroll down further for additional details as appropriate.

artemis.profile
   system properties and JVM arguments (e.g. Xmx, Xms, etc.)
artemis-roles.properties
user/role mapping for the default properties-based JAAS login module

artemis-users.properties
user/password for the default properties-based JAAS login module

bootstrap.xml
embedded web server, security, location of broker.xml

broker.xml
core broker configuration, e.g. acceptors, addresses, queues, diverts, clustering; full reference

jolokia-access.xml
security for Jolokia, specifically Cross-Origin Resource Sharing (CORS)

log4j2.properties
logging config like levels, log file locations, etc.

login.config
standard Java configuration for JAAS security

management.xml
remote connectivity and security for JMX MBeans

18.4.1. Bootstrap Configuration File
The bootstrap.xml file is very simple. Let's take a look at an example:

```xml
<broker xmlns="http://activemq.apache.org/schema">
  <jaas-security domain="activemq"/>
  <server configuration="file:/path/to/broker.xml"/>
  <web path="web">
    <binding uri="http://localhost:8161">
      <app url="activemq-branding" war="activemq-branding.war"/>
      <app url="artemis-plugin" war="artemis-plugin.war"/>
      <app url="console" war="console.war"/>
    </binding>
  </web>
</broker>
```

jaas-security
Configures JAAS-based security for the server. The domain attribute refers to the relevant login module entry in login.config. If different behavior is needed then a custom security manager can be configured by replacing jaas-security with security-manager. See the "Custom Security Manager" section in the security chapter for more details.
server
Instantiates a core server using the configuration file from the configuration attribute. This is the main broker POJO necessary to do all the real messaging work.

web
Configures an embedded web server for things like the admin console.

18.4.2. Broker configuration file

The configuration for the Apache ActiveMQ Artemis core broker is contained in broker.xml.

There are many attributes which you can configure for Apache ActiveMQ Artemis. In most cases the defaults will do fine, in fact every attribute can be defaulted which means a file with a single empty configuration element is a valid configuration file. The different configuration will be explained throughout the manual or you can refer to the configuration reference here.

18.5. Other Use-Cases

18.5.1. System Property Substitution

It is possible to use system property substitution in all the configuration files. by replacing a value with the name of a system property. Here is an example of this with a connector configuration:

```xml
<connector name="netty">
  tcp://${activemq.remoting.netty.host:localhost}:${activemq.remoting.netty.port:61616}
</connector>
```

Here you can see we have replaced 2 values with system properties activemq.remoting.netty.host and activemq.remoting.netty.port. These values will be replaced by the value found in the system property if there is one, if not they default back to localhost or 61616 respectively. It is also possible to not supply a default (i.e. ${activemq.remoting.netty.host}), however the system property must be supplied in that case.

18.5.2. Windows Server

On windows you will have the option to run ActiveMQ Artemis as a service. Just use the following command to install it:

```
$ ./artemis-service.exe install
```

The create process should give you a hint of the available commands available for the artemis-service.exe

18.5.3. Adding Bootstrap Dependencies

Bootstrap dependencies like logging handlers must be accessible by the log manager at boot time.
Package the dependency in a jar and put it on the boot classpath before of log manager jar. This can be done appending the jar at the variable JAVA_ARGS, defined in artemis.profile, with the option -Xbootclasspath/a.

- the environment variable JAVA_ARGS_APPEND can be used to append or override options.

18.5.4. Adding Runtime Dependencies

Runtime dependencies like transformers, broker plugins, JDBC drivers, password decoders, etc. must be accessible by the broker at runtime. Package the dependency in a jar, and put it on the broker's classpath. This can be done by placing the jar file in the lib directory of the broker distribution itself, by placing the jar file in the lib directory of the broker instance, by setting the system property artemis.extra.libs with the directory that contains the jar file, or by setting the environment variable ARTEMIS_EXTRA_LIBS with the directory that contains the jar file. A broker instance does not have a lib directory by default so it may need to be created. It should be on the "top" level with the bin, data, log, etc. directories. The system property artemis.extra.libs is a comma separated list of directories that contains jar files, i.e.

-Dartemis.extra.libs=/usr/local/share/java/lib1,/usr/local/share/java/lib2

The environment variable ARTEMIS_EXTRA_LIBS is a comma separated list of directories that contains jar files and is ignored if the system property artemis.extra.libs is defined, i.e.

export ARTEMIS_EXTRA_LIBS=/usr/local/share/java/lib1,/usr/local/share/java/lib2

18.5.5. Library Path

If you’re using the Asynchronous IO Journal on Linux, you need to specify java.library.path as a property on your Java options. This is done automatically in the scripts.

If you don’t specify java.library.path at your Java options then the JVM will use the environment variable LD_LIBRARY_PATH.

You will need to make sure libaio is installed on Linux. For more information refer to the libaio chapter.
Chapter 19. Command Line Interface

ActiveMQ Artemis has a Command Line Interface (CLI) that can be used to manage a few aspects of the broker like instance creation, basic user management, queue & address management, etc.

There are two ways the CLI can be used:

- Traditional CLI commands, e.g.: `.artemis [COMMAND] [PARAMETERS]`
- A custom shell that is accessed using the `.artemis` or `.artemis shell` commands.

All commands available through the traditional CLI commands are also available through the shell interface.

One benefit of the traditional CLI commands is that they can be used in your own bash scripts for automation, etc.

One benefit of the shell is that it will **reuse** some information as you repeat commands. For example, once you supply the broker URI and username & password to one command those values will be transparently applied to other commands in the same shell session. Of course, the shell also allows you to avoid retyping `.artemis` for every command.

19.1. Getting Help

You can get a complete list of available commands by typing:

```
$ ./artemis help
Usage: artemis [COMMAND]
ActiveMQ Artemis Command Line
Commands:
  help           use 'help <command>' for more information
  auto-complete  Generates the auto complete script file to be used in bash or zsh.
  shell          JLine3 shell helping using the CLI
  producer       Send message(s) to a broker.
  transfer       Move messages from one destination towards another destination.
  consumer       Consume messages from a queue.
  browser        Browse messages on a queue.
  mask           Mask a password and print it out.
  version        Print version information.
  perf           use 'help perf' for sub commands list
  check          use 'help check' for sub commands list
  queue          use 'help check' for sub commands list
  address        use 'help address' for sub commands list
  data           use 'help data' for sub commands list
  create         Create a new broker instance.
  upgrade        Update a broker instance to the current artemis.home, keeping all the data and broker.xml. Warning: backup your instance before using this command and compare the files.
```
It is also possible to use help at a specific command or sub-command for more information. For example, to get a list of sub-commands for data you type ./artemis help data:

```
$ ./artemis help data
Usage: artemis data [COMMAND]
use 'help data' for sub commands list
Commands:
  recover  Recover (undelete) every message on the journal by creating a new
           output journal. Rolled back and acked messages will be sent out to
           the output as much as possible.
  print    Print data records information. WARNING: don't use while a
           production server is running.
  exp      Export all message-data using an XML that could be interpreted by
           any system.
  imp      Import all message-data using an XML that could be interpreted by
           any system.
  decode   Decode a journal's internal format into a new set of journal files.
  encode   Encode a set of journal files into an internal encoded data format.
  compact  Compact the journal of a non running server.
```

Or you can get help for a particular command. For example, ./artemis help create:

```
Usage: artemis create [--aio] [--allow-anonymous] [--autocreate] [--autodelete]
              [--backup] [--blocking] [--clustered]
              [--disable-persistence] [--failover-on-shutdown]
              [--no-amqp-acceptor] [--no-autocreate] [--no-autotune]
              [--no-fsync] [--no-hornetq-acceptor] [--no-mqtt-acceptor]
              [--no-stomp-acceptor] [--no-web] [--paging]
              [--relax-jolokia] [--replicated] [--require-login]
              [--shared-store] [--silent] [--slave]
              [--support-advisory]
              [--suppress-internal-management-objects]
              [--use-client-auth] [--verbose] [--windows]
              [--addresses=<addresses>]
              [--cluster-password=<clusterPassword>]
              [--cluster-user=<clusterUser>] [--data=<data>]
              [--default-port=<defaultPort>] [--encoding=<encoding>]
              [--etc=<etc>] [--global-max-messages=<globalMaxMessages>]
              [--global-max-size=<globalMaxSize>] [--home=<home>]
              [--host=<host>] [--http-host=<httpHost>]
              [--http-port=<httpPort>] [--java-memory=<javaMemory>]
              [--jdbc-bindings-table-name=<jdbcBindings>]
              [--jdbc-connection-url=<jdbcURL>]
              [--jdbc-driver-class-name=<jdbcClassName>]
              [--jdbc-large-message-table-name=<jdbcLargeMessages>]
              [--jdbc-lock-expiration=<jdbcLockExpiration>]
              [--jdbc-lock-renew-period=<jdbcLockRenewPeriod>]
              [--jdbc-message-table-name=<jdbcMessages>]
```
Create a new broker instance.

The instance directory to hold the broker's configuration and data. Path must be writable.

```
<directory>
  --addresses=<addresses>
    A comma separated list of addresses with the option to specify a routing type, e.g.
    --addresses myAddress1,myAddress2:anycast.
    Routing-type default: multicast.

  --aio
    Set the journal as asyncio.

  --allow-anonymous
    Allow connections from users with no security credentials. Opposite of --require-login.
    Default: input.

  --autocreate
    Allow automatic creation of addresses & queues.
    Default: true.

  --autodelete
    Allow automatic deletion of addresses & queues.
    Default: false.

  --backup
    Be a backup broker. Valid for shared store or replication.

  --blocking
    Block producers when address becomes full.
    Opposite of --paging. Default: false.

  --cluster-password=<clusterPassword>
    The password to use for clustering. Default: input.

  --cluster-user=<clusterUser>
    The user to use for clustering. Default: input.

  --clustered
    Enable clustering.

  --data=<data>
    Directory where ActiveMQ data are stored. Paths can be absolute or relative to artemis.instance directory. Default: data.

  --default-port=<defaultPort>
    The port number to use for the main 'artemis' acceptor. Default: 61616.

  --disable-persistence
    Disable message persistence to the journal.

  --encoding=<encoding>
    The encoding that text files should use. Default: UTF-8.
```
 Directory where ActiveMQ configuration is located. Paths can be absolute or relative to artemis instance directory. Default: etc.

--failover-on-shutdown Whether broker shutdown will trigger failover for clients using the core protocol. Valid only for shared store. Default: false.

--force Overwrite configuration at destination directory.

--global-max-messages=<globalMaxMessages>
Maximum number of messages that will be accepted in memory before using address full policy mode. Default: undefined.

--global-max-size=<globalMaxSize>
Maximum amount of memory which message data may consume. Default: half of the JVM's max memory.

--home=<home>
Directory where ActiveMQ Artemis is installed.

--host=<host>
Broker's host name. Default: 0.0.0.0 or input if clustered.

--http-host=<httpHost>
Embedded web server's host name. Default: localhost.

--http-port=<httpPort>
Embedded web server's port. Default: 8161.

--java-memory=<javaMemory>
Define the -Xmx memory parameter for the broker. Default: 26.

--java-options=<javaOptions>
Extra Java options to be passed to the profile.

--jdbc Store message data in JDBC instead of local files.
--jdbc-bindings-table-name=<jdbcBindings>
Name of the jdbc bindings table.

--jdbc-connection-url=<jdbcURL>
The URL used for the database connection.

--jdbc-driver-class-name=<jdbcClassName>
JDBC driver classname.

--jdbc-large-message-table-name=<jdbcLargeMessages>
Name of the large messages table.

--jdbc-lock-expiration=<jdbcLockExpiration>
Lock expiration (in milliseconds).

--jdbc-lock-renew-period=<jdbcLockRenewPeriod>
Lock Renew Period (in milliseconds).

--jdbc-message-table-name=<jdbcMessages>
Name of the jdbc messages table.

--jdbc-network-timeout=<jdbcNetworkTimeout>
Network timeout (in milliseconds).

--jdbc-node-manager-table-name=<jdbcNodeManager>
Name of the jdbc node manager table.

--jdbc-page-store-table-name=<jdbcPageStore>
Name of the page store messages table.

--journal-device-block-size=<journalDeviceBlockSize>
The block size of the journal's storage device. Default: 4096.

--journal-retention=<retentionDays>
Configure journal retention in days. If > 0 then
enable journal-retention-directory from broker.

xml allowing replay options.

--journal-retention-max-bytes=<retentionMaxBytes>
Maximum number of bytes to keep in the retention directory.

--linux, --cygwin
Force Linux or Cygwin script creation. Default: based on your actual system.

--mapped
Set the journal as mapped.

--max-hops=<maxHops>
Number of hops on the cluster configuration.

--message-load-balancing=<messageLoadBalancing>
Message load balancing policy for cluster. Default: ON_DEMAND. Valid values: ON_DEMAND, STRICT, OFF, OFF_WITH_REDISTRIBUTION.

--name=<name>
The name of the broker. Default: same as host name.

--nio
Set the journal as nio.

--no-amqp-acceptor
Disable the AMQP specific acceptor.

--no-autocreate
Disable auto creation for addresses & queues.

--no-autotune
Disable auto tuning of the journal-buffer-timeout in broker.xml.

--no-fsync
Disable usage of fdatasync (channel.force(false) from Java NIO) on the journal.

--no-hornetq-acceptor
Disable the HornetQ specific acceptor.

--no-mqtt-acceptor
Disable the MQTT specific acceptor.

--no-stomp-acceptor
Disable the STOMP specific acceptor.

--no-web
Whether to omit the web-server definition from bootstrap.xml.

--paging
Page messages to disk when address becomes full. Opposite of --blocking. Default: true.

--password=<password>
The user's password. Default: input.

--ping=<ping>
A comma separated string to be passed on to the broker config as network-check-list. The broker will shutdown when all these addresses are unreachable.

--port-offset=<portOffset>
How much to off-set the ports of every acceptor.

--queues=<queues>
A comma separated list of queues with the option to specify a routing type, e.g. --queues myQueue1,myQueue2:multicast. Routing-type default: anycast.

--relax-jolokia
Disable strict checking in jolokia-access.xml.

--replicated
Enable broker replication.

--require-login
Require security credentials from users for connection. Opposite of --allow-anonymous.

--role=<role>
The name for the role created. Default: amq.

--security-manager=<securityManager>
Which security manager to use - jaas or basic. Default: jaas.

--shared-store
Enable broker shared store.

--silent
Disable all the inputs, and make a best guess for any required input.

--slave
Deprecated for removal. Use 'backup' instead.
19.2. Bash and Zsh auto complete

Bash and Zsh provide ways to auto-complete commands. To integrate with that functionality you have the option to generate the auto-complete script, i.e.:

```
$ ./artemis auto-complete
```

This will generate a file named `auto-complete-artemis.sh` that can be installed using:

```
$ source ./auto-complete-artemis.sh
```

After the auto-completion is installed you can view auto-completion information by pressing \( \text{TAB} \):

```
$ ./artemis
```

In order to see the various parameters available you must type -- then press \( \text{TAB} \):
19.3. Input required

Some functionality may require interactive user input if not explicitly provided through a parameter. For example, in cases like connecting to a broker or creating the broker instance:

```
$ ./artemis create
--addresses --jdbc-bindings-table-name
--paging --jdbc-connection-url
--password --jdbc-driver-class-name
--aio --jdbc-large-message-table-name
--allow-anonymous --jdbc-lock-expiration
--autocreate --jdbc-lock-renew-period
-offset --jdbc-message-table-name
--autodelete
--queues
--blocking
--relax-jolokia
--cluster-password
--replicated
--cluster-user
--require-login
--clustered
--role
```

```
$ ./artemis queue stat
Connection brokerURL = tcp://localhost:61616
Connection failed::AMQ229031: Unable to validate user from /127.0.0.1:56320. Username: null; SSL certificate subject DN: unavailable

--user:
Type the username for a retry
myUser

--password: is mandatory with this configuration:
Type the password for a retry
```

19.4. Artemis Shell

To initialize the shell session, type `./artemis shell` (or just `./artemis` if you prefer):

```
$ ./artemis
```

The ActiveMQ Artemis shell provides an interface that can be used to execute commands directly without leaving the Java Virtual Machine.
19.4.1. Connecting Interactively

It is possible to authenticate your CLI client once to the server and reuse the connection information for additional commands:

```
Apache ActiveMQ Artemis > connect --user=myUser --password=myPass --url tcp://localhost:61616
Connection brokerURL = tcp://localhost:61616
Connection Successful!
```

Now any command requiring authentication will reuse these parameters.

For example the sub-command `queue stat` will reuse previous information to perform its connection to the broker:

```
Apache ActiveMQ Artemis > queue stat
Connection brokerURL = tcp://localhost:61616
|NAME                     |ADDRESS
|CONSUMER_COUNT|MESSAGE_COUNT|MESSAGES_ADDED|DELIVERING_COUNT|MESSAGES_ACKED|SCHEDULED _COUNT|ROUTING_TYPE|
|DLQ                      |DLQ                      |0             |0            |0              |0              |ANYCAST     |
|0               |0             |0              |ANYCAST     |
|ExpriyQueue              |ExpriyQueue              |0             |0            |0              |0              |ANYCAST     |
|0               |0             |0              |ANYCAST     |
|Order                    |Order                    |0             |4347         |4347            |4347            |ANYCAST     |
|0               |0             |0              |ANYCAST     |
|activemq.management.0b...|activemq.management.0b...|1             |0            |0              |0              |MULTICAST   |
```

19.4.2. Connecting Statically

It is possible to start the shell with an initial connection configured statically, e.g.:

```
$.artemis shell --user <username> --password <password> --url tcp://<hostname>:<port>
```
The CLI should not ask for a the broker URL or user/password for any further commands, e.g.:

```
$ ./artemis shell --user myUser --password myPass
...
```

Apache ActiveMQ Artemis > queue stat
Connection brokerURL = tcp://localhost:61616

```
|NAME                     |ADDRESS |
|CONSUMER_COUNT|MESSAGE_COUNT|MESSAGES_ADDED|DELIVERING_COUNT|MESSAGES_ACKED|SCHEDULED_COUNT|ROUTING_TYPE|
|DLQ                      |DLQ                      |0             |0            |0             |0                        |ANYPACKAST |
|0               |0             |0              |ANYCAST     |                          |
|ExpriyQueue              |ExpriyQueue              |0             |0            |0             |0                        |
|0               |0             |0              |ANYCAST     |                          |
|TEST                     |TEST                     |0             |8743         |8743          |
|activemq.management.2a...|activemq.management.2a...|1             |0            |0             |
|0               |0             |0              |MULTICAST   |                          |
```
Chapter 20. The Client Classpath

20.1. Maven dependencies

The recommended way to define a client dependency for your java application is through a Maven
dependency declaration.

There are two dependencies you can choose from, org.apache.activemq:artemis-jms-client for JMS
2.0 or org.apache.activemq:artemis-jakarta-client for Jakarta Messaging 3.x.

For JMS:

```
...  
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>artemis-jms-client</artifactId>
    <version>2.34.0</version>
</dependency>
...
```

For Jakarta:

```
...  
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>artemis-jakarta-client</artifactId>
    <version>2.34.0</version>
</dependency>
...
```

20.2. Individual client dependencies

If you don’t wish to use a build tool such as Maven which manages the dependencies for you, you
may also choose to add the specific dependency jars to your classpath, which are all included under
./lib on the main distribution.

For more information of the clients individual dependencies, see:

- JMS client dependencies
- Jakarta client dependencies

20.3. Repackaged '-all' clients

Even though it is highly recommend to use the maven dependencies, in cases this isnt a possibility
and neither is using the individual dependencies as detailed above then the all-inclusive
repackaged jar could be used as an alternative.

These jars are available at Maven Central:

- artemis-jms-client-all-2.34.0.jar
- artemis-jakarta-client-all-2.34.0.jar

Whether you are using JMS or just the Core API simply add the artemis-jms-client-all jar to your client classpath. For Jakarta Messaging add the artemis-jakarta-client-all jar instead.

⚠️ These repackaged jars include all the client's dependencies. Be careful with mixing other components jars in your application as they may clash with each other. Note also that the '-all' clients cant be embedded in the same JVM as the broker, for that you must use artemis-jms-client or artemis-jakarta-client as appropriate.
Chapter 21. Address Model

Every messaging protocol and API that Apache ActiveMQ Artemis supports defines a different set of messaging resources.

- JMS uses queues and topics
- STOMP uses generic destinations
- MQTT uses topics
- AMQP uses generic nodes

In order to deal with the unique semantics and use-cases for each of these the broker has a flexible and powerful address model based on the following core set of resources:

- address
- queue
- routing type

21.1. Address

Messages are sent to an address. An address is given a unique name, a routing type, and zero or more queues.

21.2. Queue

Messages are consumed from a queue. A queue is bound to an address. It is given a unique name and a routing type. There can be zero or more queues bound to one address. When a message is sent to an address it is routed to one or more of its queues based on the configured routing type.

The name of the queue must be globally unique. For example, you can’t have a queue named q1 on address a1 and also a queue named q1 address a2.

21.3. Routing Type

A routing type determines how messages are routed from an address to the queue(s) bound to that address. Two different routing types are supported, anycast and multicast.

<table>
<thead>
<tr>
<th>If you want your messages routed to...</th>
<th>Use this routing type...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a single queue on the address</td>
<td>anycast</td>
</tr>
<tr>
<td>every queue on the address</td>
<td>multicast</td>
</tr>
</tbody>
</table>

It is possible to define queues with a different routing type for the same address, but this typically results in an anti-pattern and is therefore not recommended.
21.4. Automatic Configuration

By default Apache ActiveMQ Artemis will automatically create addresses and queues to support the semantics of whatever protocol you’re using. The broker understands how to support each protocol’s functionality with the core resources so that in most cases no manual configuration is required. This saves you from having to preconfigure each address and queue before a client can connect to it.

The broker can optionally be configured to automatically delete addresses and queues when they are no longer in use.

Automatic creation and deletion is configured on a per address basis and is controlled by the following address-setting elements:

- auto-create-addresses
- auto-delete-addresses
- default-address-routing-type
- auto-create-queues
- auto-delete-queues
- default-queue-routing-type

Automatic queue creation is for queues that would not otherwise be created during normal operation. For example, when a remote application creates a consumer on a JMS topic then a queue will be created representing that subscription as described in the JMS-to-core mapping chapter. This queue will be created regardless of how auto-create-queues is configured because it is required for normal operation.

See the documentation on address settings for more details on these elements.

Of course, automatic configuration can be disabled and everything can be configured manually. Read on for more details about manual configuration.

21.5. Basic Manual Configuration

The following examples show how to configure resources for basic anycast and multicast use-cases.

Many of the details of these use-cases are protocol agnostic. The goal here is to demonstrate and explain the basic configuration elements and how the address model works fundamentally.

21.5.1. Anycast

The most common use-case for anycast semantics, sometimes referred to as point-to-point, involves applications following a “competing consumer” pattern to receive messages from a shared queue. The more consumers receiving messages the greater the overall message throughput. Multiple Java
applications sharing a JMS queue is a classic example of this use-case.

In this use-case the broker is configured, for example, with an address, `address.foo` using the **anycast** routing type with just one queue, `q1`. When a producer sends a message to `address.foo` it is then routed to `q1` and finally dispatched to one of the consumers.

![Figure 1. Anycast](image-url)

This is what the configuration for this use-case would look like in `etc/broker.xml`:

```xml
<addresses>
  <address name="address.foo">
    <anycast>
      <queue name="q1"/>
    </anycast>
  </address>
</addresses>
```

For most protocols and APIs which support this kind of use-case (e.g. JMS, AMQP, etc.) it is customary to use the **same name** when sending and consuming messages. In that case you’d use a configuration like this:

```xml
<addresses>
  <address name="orderQueue">
    <anycast>
      <queue name="orderQueue"/>
    </anycast>
  </address>
</addresses>
```

### 21.5.2. Multicast

The most common use-case for multicast semantics, sometimes referred to as publish/subscribe or "pub/sub", involves each application receiving every message sent to an address. Multiple applications consuming from a JMS topic is a classic example of this use-case. MQTT subscriptions is another supported example of multicast semantics.

In this use-case the broker is configured with an address, `address.foo` using the **multicast** routing type with two queues, `q1` & `q2`. When a producer sends a message to `address.foo` it is then routed to
both q1 & q2 so that ultimately both consumers receive the same messages.

![Multicast Diagram]

Figure 2. Multicast

This is what the configuration for this use-case would look like in `etc/broker.xml`:

```xml
<addresses>
  <address name="address.foo">
    <multicast>
      <queue name="q1"/>
      <queue name="q2"/>
    </multicast>
  </address>
</addresses>
```

This basic configuration is simple and straight-forward, but there's a problem. In a normal pub/sub use-case like with a JMS topic or with MQTT the number of subscribers isn't known ahead of time. In that case, this is the recommended configuration:

```xml
<addresses>
  <address name="address.foo">
    <multicast/>
  </address>
</addresses>
```

Define `<multicast/>` with no queues and the broker will automatically create queues for each subscription when the consumers connect to `address.foo`. Then when a message is sent to `address.foo` it will be routed to each queue for each subscriber and therefore each subscriber will get every message. These queues are often referred to as subscription queues for obvious reasons.

These subscription queues are typically named based on the semantics of the protocol used to create them. For example, JMS supports durable and non-durable subscriptions. The queue for a non-durable subscription is named with a UUID, but the queue used for a durable subscription is named according to the JMS "client ID" and "subscription name." Similar conventions are used for AMQP, MQTT, STOMP, etc.
21.6. Advanced Manual Configuration

21.6.1. Fully Qualified Queue Names

In most cases it’s not necessary or desirable to statically configure the aforementioned subscription queues. However, there are scenarios where a user may want to statically configure a subscription queue and later connect to that queue directly using a **Fully Qualified Queue Name** (FQQN).

An FQQN uses a special syntax to specify both the address and the queue so that applications using protocols and APIs which don't natively understand the address/queue separation (e.g. AMQP, JMS, etc.) can send messages or subscribe directly to a queue rather than being limited to the address. Applications simply need to use the address name and the queue name separated by `::` (e.g. `address::queue`).

In this example, the address `a1` is configured with two queues: `q1`, `q2` as shown in the configuration below.

```xml
<addresses>
  <address name="a1">
    <multicast>
      <queue name="q1"/>
      <queue name="q2"/>
    </multicast>
  </address>
</addresses>
```

Here’s a snippet of Java code using JMS which demonstrates the FQQN syntax:

```java
Queue q1 = session.createQueue("a1::q1");
MessageConsumer consumer = session.createConsumer(q1);
```

The string `::` should only be used for FQQN and not in any other context in address or queue names.

The examples below show how to use broker side configuration to statically configure a queue with publish subscribe behavior for shared, non-shared, durable and non-durable subscription behavior.

**Shared, Durable Subscription Queue using max-consumers**

The default behavior for queues is to not limit the number connected queue consumers. The `max-consumers` parameter of the queue element can be used to limit the number of connected consumers allowed at any one time.

Open the file `etc/broker.xml` for editing.
Non-shared, Durable Subscription Queue

The broker can be configured to prevent more than one consumer from connecting to a queue at any one time. The subscriptions to queues configured this way are therefore "non-shared". To do this simply set the `<max-consumers>` parameter to 1:

```xml
<addresses>
  <address name="durable.foo">
    <multicast>
      <!-- pre-configured non shared durable subscription queue -->
      <queue name=q1 max-consumers=1>
        <durable>true</durable>
      </queue>
    </multicast>
  </address>
</addresses>
```

Non-durable Subscription Queue

Non-durable subscriptions are again usually managed by the relevant protocol manager, by creating and deleting temporary queues.

If a user requires to pre-create a queue that behaves like a non-durable subscription queue the `<purge-on-no-consumers>` flag can be enabled on the queue. When `<purge-on-no-consumers>` is set to `true`, the queue will not start receiving messages until a consumer is attached. When the last consumer is detached from the queue. The queue is purged (its messages are removed) and will not receive any more messages until a new consumer is attached.

Open the file `etc/broker.xml` for editing.

```xml
<addresses>
  <address name="non.shared.durable.foo">
    <multicast>
      <queue name=orders1 purge-on-no-consumers=true/>
    </multicast>
  </address>
</addresses>
```
21.6.2. Disabled Queue

If a user requires to statically configure a queue and disable routing to it, for example where a queue needs to be defined so a consumer can bind, but you want to disable message routing to it for the time being.

Or you need to stop message flow to the queue to allow investigation keeping the consumer bound, but don’t wish to have further messages routed to the queue to avoid message build up.

When enabled is set to true the queue will have messages routed to it. (default)

When enabled is set to false the queue will NOT have messages routed to it.

Open the file etc/broker.xml for editing.

```
<addresses>
  <address name="foo.bar">
    <multicast>
      <queue name="orders1" enabled="false"/>
    </multicast>
  </address>
</addresses>
```

Disabling all the queues on an address means that any message sent to that address will be silently dropped.

21.6.3. Temporary Queues

For some protocols and APIs which only support monolithic "destinations" without the address/queue separation (e.g. AMQP, JMS, etc.) temporary queues are created by the broker using a UUID (i.e universally unique identifier) as the name for both the address and the queue. Because the name is a UUID it is impossible to create an address-setting for it whose match is anything but #.

To solve this problem one can specify the temporary-queue-namespace in broker.xml and then create an address-setting whose match value corresponds to the configured temporary-queue-namespace. When the temporary-queue-namespace is set and a temporary queue is created then the broker will prepend the temporary-queue-namespace value along with the delimiter value configured in wildcard-addresses (defaults to .) to the address name and use that to lookup the associated address-setting values.

Here’s a simple example configuration:

```
<temporary-queue-namespace>temp</temporary-queue-namespace>

<address-settings>
  <address-setting match="temp.#"/>
</address-settings>
```
Using this configuration any temporary queue will have metrics disabled.

This setting does not change the actual name of the temporary queue. It only changes the name used to lookup the address-settings.

21.6.4. Other Advanced Configurations

Each of the following advanced configurations have their own chapter so their details are not repeated here:

- Exclusive queues
- Last Value queues
- Non-Destructive queues
- Ring queues
- Retroactive addresses

21.7. How to filter messages

Apache ActiveMQ Artemis supports the ability to filter messages using Filter Expressions.

Filters can be applied in two places - on a queue and on a consumer.

Filtering messages on a queue increases performance vs. filtering on the consumer because the messages don't need to be scanned. However, a queue filter is often not as flexible.

21.7.1. Queue Filter

When a filter is applied to a queue, messages are filtered before they are routed to the queue. To add a filter use the filter element when configuring a queue, e.g.:

```xml
<addresses>
  <address name="filter">
    <anycast>
      <queue name="filter">
        <filter string="color='red'"/>
      </queue>
    </anycast>
  </address>
</addresses>
```

The filter defined above ensures that only messages with an attribute "color='red'" is sent to this queue.
21.7.2. Consumer Filters

Consumer filters are applied after messages have routed to the queue and are defined using the appropriate client APIs. The following JMS example shows how consumer filters work.

Define an address with a single queue, with no filter applied in etc/broker.xml.

```
<addresses>
  <address name="filter">
    <anycast>
      <queue name="filter"/>
    </anycast>
  </address>
</addresses>
```

Then send some messages to the queue.

```
...
// Send some messages
for (int i = 0; i < 3; i++) {
  TextMessage redMessage = senderSession.createTextMessage("Red");
  redMessage.setStringProperty("color", "red");
  producer.send(redMessage)

  TextMessage greenMessage = senderSession.createTextMessage("Green");
  greenMessage.setStringProperty("color", "green");
  producer.send(greenMessage)
}
```

At this point the queue would have 6 messages: red, green, red, green, red, green.

Create a consumer with the filter color='red'.

```
MessageConsumer redConsumer = redSession.createConsumer(queue, "color='red'");
```

The redConsumer has a filter that only matches "red" messages. The redConsumer will receive 3 messages.

red, red, red

The resulting queue would now be

green, green, green
21.8. Alternate Ways to Determine Routing Type

Typically the routing type is determined either by the static XML configuration or by the default-address-routing-type and default-queue-routing-type address-setting elements used for automatic address and queue creation. However, there are two other ways to specify routing type:

- a configurable prefix which client applications can use when sending messages or creating consumers
- a property client applications can set on the messages they send

21.8.1. Using a Prefix to Determine Routing Type

These prefixes are configured using the anycastPrefix and multicastPrefix parameters within the URL of the acceptor which the client is using. When multiple values are needed, these can be separated by a comma.

Configuring an Anycast Prefix

In etc/broker.xml, add the anycastPrefix to the URL of the desired acceptor. In the example below, the acceptor is configured to use queue/ for the anycastPrefix. Client code can specify queue/foo/ if the client wants anycast routing.

```
<acceptor name="artemis">
tcp://0.0.0.0:61616?protocols=AMQP;anycastPrefix=queue/</acceptor>
```

Consider, for example, a STOMP client that wants to send a message using anycast semantics to a queue that doesn’t exist. Consider also that the broker is configured to auto-create addresses and queues, but the default-address-routing-type and default-queue-routing-type are both MULTICAST. Since the anycastPrefix is queue/ it can just send a message to queue/foo and the broker will automatically create an address named foo with an anycast queue also named foo.

Configuring a Multicast Prefix

In etc/broker.xml, add the multicastPrefix to the URL of the desired acceptor. In the example below, the acceptor is configured to use topic/ for the multicastPrefix. Client code can specify topic/foo/ if the client wants multicast routing.

```
<acceptor name="artemis">
tcp://0.0.0.0:61616?protocols=AMQP;multicastPrefix=topic/</acceptor>
```

Consider, for example, a STOMP client that wants to create a subscription with multicast semantics on an address that doesn’t exist. Consider also that the broker is configured to auto-create addresses and queues, but the default-address-routing-type and default-queue-routing-type are both ANYCAST. Since the multicastPrefix is topic/ it can just subscribe to topic/foo and the broker will automatically create an address named foo with a multicast queue for the subscription. Any messages sent to foo will then be routed to the subscription queue.
21.8.2. Using a Message Property to Determine Routing Type

The `AMQ_ROUTING_TYPE` property represents a byte value which will be used by the broker to determine the routing type when a message is sent. Use 0 for anycast routing or 1 for multicast routing.

A message will only be routed to queues which match its `AMQ_ROUTING_TYPE` property value (if any). For example, if a message with an `AMQ_ROUTING_TYPE` value of 1 (i.e. multicast) is sent to an address that only has anycast queues then the message won't actually be routed to any of the queues since the routing types don't match. If no `AMQ_ROUTING_TYPE` is set then the message will be routed to all the queues on the address according to the queues' routing semantics.
Chapter 22. Address Settings

With address settings you can provide a block of settings which will be applied to any addresses that match the string in the `match` attribute. In the below example the settings would only be applied to the address `order.foo` address, but it is also possible to use wildcards to apply settings.

For example, if you used the `match` string `queue.#` the settings would be applied to all addresses which start with `queue`.

Address settings are **hierarchical**. Therefore, if more than one `address-setting` would match then the settings are applied in order of their specificity with the more specific match taking priority. A match on the any-words delimiter (# by default) is considered less specific than a match without it. A match with a single word delimiter(*) by default) is considered less specific than a match on an exact queue name. In this way settings can be "layered" so that configuration details don't need to be repeated.

Address setting matches can also be "literal" which can be used to match wildcards literally, for further details see [literal matches](#).

The meaning of the specific settings are explained fully throughout the user manual, however here is a brief description with a link to the appropriate chapter if available.

Here an example of an `address-setting` entry that might be found in the `broker.xml` file.

```xml
<address-settings>
  <address-setting match="order.foo">
    <dead-letter-address>DLA</dead-letter-address>
    <auto-create-dead-letter-resources>false</auto-create-dead-letter-resources>
    <dead-letter-queue-prefix/></dead-letter-queue-prefix>
    <dead-letter-queue-suffix></dead-letter-queue-suffix>
    <expiry-address>ExpiryQueue</expiry-address>
    <auto-create-expiry-resources>false</auto-create-expiry-resources>
    <expiry-queue-prefix></expiry-queue-prefix>
    <expiry-queue-suffix></expiry-queue-suffix>
    <expiry-delay>123</expiry-delay>
    <redelivery-delay>5000</redelivery-delay>
    <redelivery-delay-multiplier>1.0</redelivery-delay-multiplier>
    <redelivery-collision-avoidance-factor>0.0</redelivery-collision-avoidance-factor>
    <max-redelivery-delay>10000</max-redelivery-delay>
    <max-delivery-attempts>3</max-delivery-attempts>
    <max-size-bytes>-1</max-size-bytes>
    <max-size-messages>-1</max-size-messages>
    <max-size-bytes-reject-threshold>-1</max-size-bytes-reject-threshold>
    <page-size-bytes>10MB</page-size-bytes>
    <address-full-policy>PAGE</address-full-policy>
    <message-counter-history-day-limit></message-counter-history-day-limit>
    <last-value-queue>false</last-value-queue>  <!-- deprecated! see default-last-value-queue -->
  </address-setting>
</address-settings>
```
**dead-letter-address**

Is the address to which messages are sent when they exceed max-delivery-attempts. If no address is defined here then such messages will simply be discarded. Read more about undelivered messages.

**auto-create-dead-letter-resources**

Whether the broker will automatically create the defined dead-letter-address and a corresponding dead-letter queue when a message is undeliverable. Read more in the chapter
about undelivered messages.

**dead-letter-queue-prefix**

The prefix used for automatically created dead-letter queues. Default is empty. Read more in the chapter about undelivered messages.

**dead-letter-queue-suffix**

The suffix used for automatically created dead-letter queues. Default is empty. Read more in the chapter about undelivered messages.

**expiry-address**

Where to send a message that has expired. If no address is defined here then such messages will simply be discarded. Read more about message expiry.

**auto-create-expiry-resources**

Determines whether or not the broker will automatically create the defined expiry-address and a corresponding expiry queue when a message expired. Read more in the chapter about undelivered messages.

**expiry-queue-prefix**

The prefix used for automatically created expiry queues. Default is empty. Read more in the chapter about message expiry.

**expiry-queue-suffix**

The suffix used for automatically created expiry queues. Default is empty. Read more in the chapter about message expiry.

**expiry-delay**

The expiration time that will be used for messages which are using the default expiration time (i.e. 0). For example, if expiry-delay is set to "10" and a message which is using the default expiration time (i.e. 0) arrives then its expiration time of "0" will be changed to "10." However, if a message which is using an expiration time of "20" arrives then its expiration time will remain unchanged. Setting expiry-delay to "-1" will disable this feature. The default is "-1". Read more about message expiry.

**max-delivery-attempts**

defines how many time a cancelled message can be redelivered before sending to the dead-letter-address. Read more about undelivered messages.

**redelivery-delay**

defines how long to wait before attempting redelivery of a cancelled message. Default is 0. Read more about undelivered messages.

**redelivery-delay-multiplier**

The number by which the redelivery-delay will be multiplied on each subsequent redelivery attempt. Default is 1.0. Read more about undelivered messages.
redelivery-collision-avoidance-factor
defines an additional factor used to calculate an adjustment to the redelivery-delay (up or down). Default is 0.0. Valid values are between 0.0 and 1.0. Read more about undelivered messages.

max-size-bytes
max-size-messages
page-size-bytes
max-read-page-messages
max-read-page-bytes
All these are used to configure paging on an address. This is explained in the paging documentation.

max-size-bytes-reject-threshold
is used with the address full BLOCK policy, the maximum size (in bytes) an address can reach before messages start getting rejected. Works in combination with max-size-bytes for AMQP clients only. Default is -1 (i.e. no limit).

address-full-policy
This attribute can have one of the following values: PAGE, DROP, FAIL or BLOCK and determines what happens when an address where max-size-bytes is specified becomes full. The default value is PAGE. If the value is PAGE then further messages will be paged to disk. If the value is DROP then further messages will be silently dropped. If the value is FAIL then further messages will be dropped and an exception will be thrown on the client-side. If the value is BLOCK then client message producers will block when they try and send further messages. See the Flow Control and Paging chapters for more info.

message-counter-history-day-limit
is the number of days to keep message counter history for this address assuming that message-counter-enabled is true. Default is 0.

default-last-value-queue
Whether a queue only uses last values or not. Default is false. This value can be overridden at the queue level using the last-value boolean. Read more about last value queues.

default-exclusive-queue
Whether a queue will serve only a single consumer. Default is false. This value can be overridden at the queue level using the exclusive boolean. Read more about exclusive queues.

default-consumers-before-dispatch
The number of consumers needed on a queue bound to the matching address before messages will be dispatched to those consumers. Default is 0. This value can be overridden at the queue level using the consumers-before-dispatch boolean. This behavior can be tuned using delay-before-dispatch on the queue itself or by using the default-delay-before-dispatch address-setting.
default-delay-before-dispatch
The number of milliseconds the broker will wait for the configured number of consumers to connect to the matching queue before it will begin to dispatch messages. Default is -1 (wait forever).

redistribution-delay
How long to wait when the last consumer is closed on a queue before redistributing any messages. Default is -1. Read more about clusters.

send-to-dla-on-no-route
If a message is sent to an address, but the server does not route it to any queues (e.g. there might be no queues bound to that address, or none of the queues have filters that match) then normally that message would be discarded. However, if this parameter is true then such a message will instead be sent to the dead-letter-address (DLA) for that address, if it exists. Default is false.

slow-consumer-threshold
The minimum rate of message consumption allowed before a consumer is considered "slow." Measured in units specified by the slow-consumer-threshold-measurement-unit configuration option. Default is -1 (i.e. disabled); any other value must be greater than 0 to ensure a queue has messages, and it is the actual consumer that is slow. A value of 0 will allow a consumer with no messages pending to be considered slow. Read more about slow consumers.

slow-consumer-threshold-measurement-unit
The units used to measure the slow-consumer-threshold. Valid options are:

- MESSAGES_PER_SECOND
- MESSAGES_PER_MINUTE
- MESSAGES_PER_HOUR
- MESSAGES_PER_DAY

If no unit is specified the default MESSAGES_PER_SECOND will be used. Read more about slow consumers.

slow-consumer-policy
What should happen when a slow consumer is detected. KILL will kill the consumer's connection (which will obviously impact any other client threads using that same connection). NOTIFY will send a CONSUMER_SLOW management notification which an application could receive and take action with. Read more about slow consumers.

slow-consumer-check-period
How often to check for slow consumers on a particular queue. Measured in seconds. Default is 5.

This should be at least 2x the maximum time it takes a consumer to process 1 message. For example, if the slow-consumer-threshold is set to 1 and the slow-consumer-threshold-measurement-unit is set to MESSAGES_PER_MINUTE then this should be set to at least 2 x 60s i.e. 120s. Read more about slow consumers.
auto-create-queues
Whether or not the broker should automatically create a queue when a message is sent or a consumer tries to connect to a queue whose name fits the address match. Queues which are auto-created are durable, non-temporary, and non-transient. Default is true.

Automatic queue creation does not work for the core client. The core API is a low-level API and is not meant to have such automation.

Automatic queue creation is for queues that would not otherwise be created during normal operation. For example, when a remote application creates a consumer on a JMS topic then a queue will be created representing that subscription as described in the JMS-to-core mapping chapter. This queue will be created regardless of how auto-create-queues is configured because it is required for normal operation.

auto-delete-queues
Whether or not the broker should automatically delete auto-created queues when they have both 0 consumers and the message count is less than or equal to auto-delete-queues-message-count. Default is true.

auto-delete-created-queues
Whether or not the broker should automatically delete created queues when they have both 0 consumers and the message count is less than or equal to auto-delete-queues-message-count. Default is false.

auto-delete-queues-delay
How long to wait (in milliseconds) before deleting auto-created queues after the queue has 0 consumers and the message count is less than or equal to auto-delete-queues-message-count. Default is 0 (delete immediately). The broker’s address-queue-scan-period controls how often (in milliseconds) queues are scanned for potential deletion. Use -1 to disable scanning. The default scan value is 30000.

auto-delete-queues-message-count
The message count that the queue must be less than or equal to before deleting auto-created queues. To disable message count check -1 can be set. Default is 0 (empty queue).

auto-delete-queues-skip-usage-check
A queue will only be auto-deleted by default if it has actually been "used." A queue is considered "used" if any messages have been sent to it or any consumers have connected to it during its life. However, there are use-cases where it’s useful to skip this check. When set to true it is imperative to also set auto-delete-queues-delay to a value greater than 0 otherwise queues may be deleted almost immediately after being created. In this case the queue will be deleted based on when it was created rather then when it was last "used." Default is false.

the above auto-delete address settings can also be configured individually at the queue level when a client auto creates the queue.
For Core API it is exposed in createQueue methods.

For Core JMS you can set it using the destination queue attributes `my.destination?auto-delete=true&auto-delete-delay=120000&auto-delete-message-count=-1`

**config-delete-queues**

How the broker should handle queues deleted on config reload, by delete policy: **OFF** or **FORCE**. Default is **OFF**. Read more about **configuration reload**.

**config-delete-diverts**

How the broker should handle diverts deleted on config reload, by delete policy: **OFF** or **FORCE**. Default is **OFF**. Read more about **configuration reload**.

**auto-create-addresses**

Whether or not the broker should automatically create an address when a message is sent to or a consumer tries to consume from a queue which is mapped to an address whose name fits the address **match**. Default is **true**.

Automatic address creation does not work for the core client. The core API is a low-level API and is not meant to have such automation.

**auto-delete-addresses**

Whether or not the broker should automatically delete auto-created addresses once the address no longer has any queues. Default is **true**.

**auto-delete-addresses-delay**

How long to wait (in milliseconds) before deleting auto-created addresses after they no longer have any queues. Default is **0** (delete immediately). The broker's **address-queue-scan-period** controls how often (in milliseconds) addresses are scanned for potential deletion. Use **-1** to disable scanning. The default scan value is **30000**.

**auto-delete-addresses-skip-usage-check**

An address will only be auto-deleted by default if it has actually been "used." An address is considered "used" if any queues have been created on it during its life. However, there are use-cases where it's useful to skip this check. When set to **true** it is **imperative** to also set **auto-delete-addresses-delay** to a value greater than **0** otherwise addresses may be deleted almost immediately after being created. In this case the address will be deleted based on when it was created rather then when it was last "used." Default is **false**.

**config-delete-addresses**

How the broker should handle addresses deleted on config reload, by delete policy: **OFF** or **FORCE**. Default is **OFF**. Read more about **configuration reload**.

**management-browse-page-size**

is the number of messages a management resource can browse. This is relevant for the **browse**, **list** and **count-with-filter** management methods exposed on the queue control. Default is **200**.
management-message-attribute-size-limit

is the number of bytes collected from the message for browse. This is relevant for the browse and list management methods exposed on the queue control. Message attributes longer than this value appear truncated. Default is 256. Use -1 to switch this limit off. Note that memory needs to be allocated for all messages that are visible at a given moment. Setting this value too high may impact the browser stability due to the large amount of memory that may be required to browse through many messages.

default-purge-on-no-consumers

defines a queue's default purge-on-no-consumers setting if none is provided on the queue itself. Default is false. This value can be overridden at the queue level using the purge-on-no-consumers boolean. Read more about this functionality.

default-max-consumers

defines a queue's default max-consumers setting if none is provided on the queue itself. Default is -1 (i.e. no limit). This value can be overridden at the queue level using the max-consumers boolean. Read more about this functionality.

default-queue-routing-type

The routing-type for an auto-created queue if the broker is unable to determine the routing-type based on the client and/or protocol semantics. Default is MULTICAST. Read more about routing types.

default-address-routing-type

The routing-type for an auto-created address if the broker is unable to determine the routing-type based on the client and/or protocol semantics. Default is MULTICAST. Read more about routing types.

default-consumer-window-size

The default consumerWindowSize value for a CORE protocol consumer, if not defined the default will be set to 1 MiB (1024 * 1024 bytes). The consumer will use this value as the window size if the value is not set on the client. Read more about flow control.

default-ring-size

The default ring-size value for any matching queue which doesn't have ring-size explicitly defined. If not defined the default will be set to -1. Read more about ring queues.

retroactive-message-count

The number of messages to preserve for future queues created on the matching address. Defaults to 0. Read more about retroactive addresses.

enable-metrics

determines whether or not metrics will be published to any configured metrics plugin for the matching address. Default is true. Read more about metrics.

enable-ingress-timestamp

determines whether or not the broker will add its time to messages sent to the matching address. When true the exact behavior will depend on the specific protocol in use. For AMQP messages
the broker will add a long message annotation named x-opt-ingress-time. For core messages (used by the core and OpenWire protocols) the broker will add a long property named _AMQ_INGRESS_TIMESTAMP. For STOMP messages the broker will add a frame header named ingress-timestamp. The value will be the number of milliseconds since the epoch. Default is false.

id-cache-size

defines the maximum size of the duplicate ID cache for an address, as each address has its own cache that helps to detect and prevent the processing of duplicate messages based on their unique identification. By default, the id-cache-size setting inherits from the global id-cache-size, with a default of 20000 elements if not explicitly configured. Read more about duplicate id cache sizes.

22.1. Literal Matches

A literal match is a match that contains wildcards but should be applied without regard to those wildcards. In other words, the wildcards should be ignored and the address settings should only be applied to the literal (i.e. exact) match.

This can be useful when an application uses a wildcard address. For example, if an application creates a multicast queue on the address orders.#{ and that queue needs a different configuration than other matching addresses like orders.retail and orders.wholesale. Generally speaking this kind of use-case is rare, but wildcard addresses are often used by MQTT clients, and this kind of configuration flexibility is useful.

22.1.1. Configuring a Literal Match

If you want to configure a literal match the first thing to do is to configure the literal-match-markers parameter in broker.xml. This defines the beginning and ending characters used to mark the literal match, e.g.:

```
<core>
    ...
    <literal-match-markers>()[/literal-match-markers>
    ...
</core>
```

By default, no value is defined for literal-match-markers which means that literal matches are disabled by default. The value must be only 2 characters.

Once literal-match-markers is defined you can then use those markers in the match of the address setting, e.g.

```
<address-settings>
    <address-setting match="(orders.#)">
        <enable-metrics>true</enable-metrics>
    </address-setting>
    <address-setting match="orders.#">
```


Using these settings metrics will be enabled on the address `orders.#` and any queues bound directly on that address, but metrics will *not* be enabled for other matching addresses like `orders.retail` or `orders.wholesale` and any queues bound to those addresses.
Chapter 23. Wildcard Syntax

Apache ActiveMQ Artemis uses a specific syntax for representing wildcards in security settings, address settings and when creating consumers.

The syntax is similar to that used by AMQP.

An Apache ActiveMQ Artemis wildcard expression contains **words separated by a delimiter**. The default delimiter is . (full stop).

The special characters # and * also have special meaning and can take the place of a word.

To be clear, the wildcard characters cannot be used like wildcards in a regular expression. They operate exclusively on **words separated by a delimiter**.

### 23.1. Matching Any Word

The character # means "match any sequence of zero or more words".

So the wildcard `news.europe.#` would match:

- `news.europe`
- `news.europe.sport`
- `news.europe.politics`
- `news.europe.politics.regional`

But `news.europe.#` would **not** match:

- `news.usa`
- `news.usa.sport`
- `entertainment`

### 23.2. Matching a Single Word

The character * means "match a single word".

The wildcard `news.*` would match:

- `news.europe`
- `news.usa`

But `news.*` would **not** match:

- `news.europe.sport`
- `news.usa.sport`
- `news.europe.politics.regional`
The wildcard `news.*.sport` would match:

- `news.europe.sport`
- `news.usa.sport`

But `news.*.sport` would not match:

- `news.europe.politics`

### 23.3. Customizing the Syntax

It’s possible to further configure the syntax of the wildcard addresses using the broker configuration. For that, the `<wildcard-addresses>` configuration tag is used.

```xml
<wildcard-addresses>
  <routing-enabled>true</routing-enabled>
  <delimiter>.</delimiter>
  <any-words>#</any-words>
  <single-word>*</single-word>
</wildcard-addresses>
```

The example above shows the default configuration.
Chapter 24. Routing Messages With Wild Cards

Apache ActiveMQ Artemis allows the routing of messages via wildcard addresses.

If a queue is created with an address of say `queue.news.#` then it will receive any messages sent to addresses that match this, for instance `queue.news.europe` or `queue.news.usa` or `queue.news.usa.sport`. If you create a consumer on this queue, this allows a consumer to consume messages which are sent to a hierarchy of addresses.

In JMS terminology this allows "topic hierarchies" to be created.

This functionality is enabled by default. To turn it off add the following to the `broker.xml` configuration.

```xml
<wildcard-addresses>
  <routing-enabled>false</routing-enabled>
</wildcard-addresses>
```

For more information on the wild card syntax and how to configure it, take a look at wildcard syntax chapter, also see the topic hierarchy example in the examples.
Chapter 25. Diverting and Splitting Message Flows

Apache ActiveMQ Artemis allows you to configure objects called *diverts* with some simple server configuration. Diverts allow you to transparently divert messages routed to one address to one or more other addresses, without making any changes to any client application logic.

Diverts can be **exclusive** or **non-exclusive**.

An *exclusive* divert routes messages the new address(es) only. Messages are not routed to the old address at all.

A *non-exclusive* divert routes messages to the old address and a *copy* of the messages are also sent to the new address(es). Think of non-exclusive divert as splitting message flow, e.g. there may be a requirement to monitor every order sent to an order queue.

Multiple diverts can be configured for a single address. When an address has both exclusive and non-exclusive diverts configured, the exclusive ones are processed first. If any of the exclusive diverts diverted the message, the non-exclusive ones are not processed.

Diverts can also be configured to have an optional message filter. If specified then only messages that match the filter will be diverted.

Diverts can apply a particular routing-type to the message, strip the existing routing type, or simply pass the existing routing-type through. This is useful in situations where the message may have its routing-type set but you want to divert it to an address using a different routing-type. It’s important to keep in mind that a message with the *anycast* routing-type will not actually be routed to queues using *multicast* and vice-versa. By configuring the *routing-type* of the divert you have the flexibility to deal with any situation. Valid values are *ANycast*, *MULTICAST*, *PASS*, & *STRIP*. The default is *STRIP*.

Diverts can also be configured to apply a **Transformer**. If specified, all diverted messages will have the opportunity of being transformed by the **Transformer**. When an address has multiple diverts configured, all of them receive the same, original message. This means that the results of a transformer on a message are not directly available for other diverts or their filters on the same address. See the documentation on adding runtime dependencies to understand how to make your transformer available to the broker.

A divert will only divert a message to an address on the *same server*. If you want to divert to an address on a different server a common pattern would be to divert to a local "store-and-forward" queue and then set up a **bridge** which consumes from that queue and forwards to an address on a different server.

Diverts are therefore a very sophisticated concept which when combined with bridges can be used to create interesting and complex routings. The set of diverts on a server can be thought of as a type of routing table for messages. Combining diverts with bridges allows you to create a distributed network of reliable routing connections between multiple geographically distributed servers, creating your global messaging mesh.
Diverts are defined as xml in the `broker.xml` file at the `core` attribute level. There can be zero or more diverts in the file.

Diverted messages get special properties.

Please see the examples for a full working example at `./examples/features/standard/divert/` showing you how to configure and use diverts.

Let's take a look at some divert examples...

### 25.1. Exclusive Divert

Let's take a look at an exclusive divert. An exclusive divert diverts all matching messages that are routed to the old address to the new address. Matching messages do not get routed to the old address.

Here's some example xml configuration for an exclusive divert, it's taken from the divert example:

```xml
<divert name="prices-divert">
  <address>priceUpdates</address>
  <forwarding-address>priceForwarding</forwarding-address>
  <filter string="office='New York'"/>
  <transformer-class-name>
    org.apache.activemq.artemis.jms.example.AddForwardingTimeTransformer
  </transformer-class-name>
  <exclusive>true</exclusive>
</divert>
```

We define a divert called `prices-divert` that will divert any messages sent to the address `priceUpdates` to another local address `priceForwarding`.

We also specify a message filter string so only messages with the message property `office` with value `New York` will get diverted, all other messages will continue to be routed to the normal address. The filter string is optional, if not specified then all messages will be considered matched.

In this example a transformer class is specified without any configuration properties. Again this is optional, and if specified the transformer will be executed for each matching message. This allows you to change the messages body or properties before it is diverted. In this example the transformer simply adds a header that records the time the divert happened. See the transformer chapter for more details about transformer-specific configuration.

This example is actually diverting messages to a local store and forward queue, which is configured with a bridge which forwards the message to an address on another ActiveMQ Artemis server. Please see the example for more details.

### 25.2. Non-exclusive Divert

Now we'll take a look at a non-exclusive divert. Non exclusive diverts are the same as exclusive
diverts, but they only forward a *copy* of the message to the new address. The original message continues to the old address.

You can therefore think of non-exclusive diverts as *splitting* a message flow.

Non exclusive diverts can be configured in the same way as exclusive diverts with an optional filter and transformer, here's an example non-exclusive divert, again from the divert example:

```xml
<divert name="order-divert">
    <address>orders</address>
    <forwarding-address>spyTopic</forwarding-address>
    <exclusive>false</exclusive>
</divert>
```

The above divert example takes a copy of every message sent to the address *orders* and sends it to a local address called *spyTopic*.

### 25.3. Composite Divert

A *composite* divert is one which forwards messages to multiple addresses. This pattern is sometimes referred to as *fan-out*. Configuration is simple. Just use a comma separated list in *forwarding-address*, e.g.:

```xml
<divert name="shipping-divert">
    <address>shipping</address>
    <forwarding-address>dallas, chicago, denver</forwarding-address>
    <exclusive>false</exclusive>
</divert>
```
Chapter 26. Transformers

A transformer, as the name suggests, is a component which transforms a message. For example, a transformer could modify the body of a message or add or remove properties. Both diverts and core bridges support.

A transformer is simply a class which implements the interface org.apache.activemq.artemis.core.server.transformer.Transformer:

```java
public interface Transformer {
    default void init(Map<String, String> properties) {}
    Message transform(Message message);
}
```

The init method is called immediately after the broker instantiates the class. There is a default method implementation so implementing init is optional. However, if the transformer needs any configuration properties it should implement init and the broker will pass the configured key/value pairs to the transformer using a java.util.Map.

26.1. Configuration

The most basic configuration requires only specifying the transformer’s class name, e.g.:

```xml
<transformer-class-name>
    org.foo.MyTransformer
</transformer-class-name>
```

However, if the transformer needs any configuration properties those can be specified using a slightly different syntax, e.g.:

```xml
<transformer>
    <class-name>org.foo.MyTransformerWithProperties</class-name>
    <property key="transformerKey1" value="transformerValue1"/>
    <property key="transformerKey2" value="transformerValue2"/>
</transformer>
```

Any transformer implementation needs to be added to the broker’s classpath. See the documentation on adding runtime dependencies to understand how to make your transformer available to the broker.
Chapter 27. Filter Expressions

Apache ActiveMQ Artemis provides a powerful filter language based on a subset of the SQL 92 expression syntax.

It is the same as the syntax used for JMS & Jakarta Messaging selectors, but the predefined identifiers are different. For documentation on JMS selector syntax please the JavaDoc for \texttt{javax.jms.Message}. For the corresponding Jakarta Messaging JavaDoc see \texttt{jakarta.jms.Message}.

Filter expressions are used in several places in Apache ActiveMQ Artemis.

- **Predefined Queues.** When pre-defining a queue, in \texttt{broker.xml} in either the core or jms configuration a filter expression can be defined for a queue. Only messages that match the filter expression will enter the queue.

- **Core bridges** can be defined with an optional filter expression, only matching messages will be bridged (see \texttt{Core Bridges}).

- **Diverts** can be defined with an optional filter expression, only matching messages will be diverted (see \texttt{Diverts}).

- **Filter** are also used programmatically when creating consumers, queues and in several places as described in \texttt{management}.

There are some differences between JMS selector expressions and Apache ActiveMQ Artemis core filter expressions. Whereas JMS selector expressions operate on a JMS message, Apache ActiveMQ Artemis core filter expressions operate on a core message.

The following identifiers can be used in a core filter expressions to refer to attributes of the core message in an expression:

- **AMQUserID**
  
  The ID set by the user when the message is sent. This is analogous to the \texttt{JMSMessageID} for JMS-based clients.

- **AMQAddress**
  
  The address to which the message was sent.

- **AMQGroupID**
  
  The group ID used when sending the message.

- **AMQPriority**
  
  To refer to the priority of a message. Message priorities are integers with valid values from \texttt{0 - 9}. \texttt{0} is the lowest priority and \texttt{9} is the highest. e.g. \texttt{AMQPriority = 3 AND animal = 'aardvark'}

- **AMQExpiration**
  
  To refer to the expiration time of a message. The value is a long integer.

- **AMQDurable**
  
  To refer to whether a message is durable or not. The value is a string with valid values: \texttt{DURABLE}
AMQTimestamp

The timestamp of when the message was created. The value is a long integer.

AMQSize

The size of a message in bytes. The value is an integer.

Any other identifiers used in core filter expressions will be assumed to be properties of the message.

The JMS and Jakarta Messaging specs state that a String property should not get converted to a numeric when used in a selector. So for example, if a message has the age property set to String 21 then the following selector should not match it: age > 18. Since Apache ActiveMQ Artemis supports STOMP clients which can only send messages with string properties, that restriction is a bit limiting. Therefore, if you want your filter expressions to auto-convert String properties to the appropriate number type, just prefix it with convert_string_expressions:. If you changed the filter expression in the previous example to be convert_string_expressions:age > 18, then it would match the aforementioned message.

The JMS and Jakarta Messaging specs also state that property identifiers (and therefore the identifiers which are valid for use in a filter expression) are an:

unlimited-length sequence of letters and digits, the first of which must be a letter. A letter is any character for which the method Character.isJavaLetter returns true. This includes _ and $. A letter or digit is any character for which the method Character.isJavaLetterOrDigit returns true.

This constraint means that hyphens (i.e. -) cannot be used. However, this constraint can be overcome by using the hyphenated_props: prefix. For example, if a message had the foo-bar property set to 0 then the filter expression hyphenated_props:foo-bar = 0 would match it.

27.1. XPath

Apache ActiveMQ Artemis also supports special XPath filters which operate on the body of a message. The body must be XML. To use an XPath filter use this syntax:

```
XPATH '<xpath-expression>'
```

XPath filters are supported with and between producers and consumers using the following protocols:

- OpenWire JMS
- Core (and Core JMS)
- STOMP
• AMQP

Since XPath applies to the body of the message and requires parsing of XML it may be significantly slower than normal filters.

Large messages are not supported.

The XML parser used for XPath is configured with these default “features”:

- http://xml.org/sax/features/external-general-entities: false
- http://apache.org/xml/features/disallow-doctype-decl: true

However, in order to deal with any implementation-specific issues the features can be customized by using system properties starting with the org.apache.activemq.documentBuilderFactory.feature: prefix, e.g.:

Chapter 28. Management

Apache ActiveMQ Artemis has an extensive management API that allows a user to modify a server configuration, create new resources (e.g. addresses and queues), inspect these resources (e.g. how many messages are currently held in a queue) and interact with it (e.g. to remove messages from a queue). Apache ActiveMQ Artemis also allows clients to subscribe to management notifications.

There are numerous ways to access Apache ActiveMQ Artemis management API:

- Using JMX — JMX is the standard way to manage Java applications
- Using Jolokia — Jolokia exposes the JMX API of an application through an HTTP interface
- Using the Core Client — management operations are sent to Apache ActiveMQ Artemis server using Core Client messages
- Using any JMS Client — management operations are sent to Apache ActiveMQ Artemis server using JMS Client messages
- Web Console — a web application which provides a graphical interface to the management API.

Although there are four different ways to manage Apache ActiveMQ Artemis, each API supports the same functionality. If it is possible to manage a resource using JMX it is also possible to achieve the same result using Core messages.

Besides these four management interfaces, a Web Console and a Command Line management utility are also available to administrators of ActiveMQ Artemis.

The choice depends on your requirements, your application settings, and your environment to decide which way suits you best.

28.1. The Management API

Regardless of the way you invoke management operations, the management API is the same.

For each managed resource, there exists a Java interface describing what operations can be invoked for this type of resource.

To learn about available management operations, see the Javadoc for these interfaces. They are located in the org.apache.activemq.artemis.api.core.management package and they are named with the word Control at the end.

28.1.1. Server Management

The ActiveMQServerControl interface is the entry point for broker management.

- Listing, creating, deploying and destroying queues
  
  A list of deployed queues can be retrieved using the getQueueNames() method.
  
  Queues can be created or destroyed using the management operations createQueue() or deployQueue() or destroyQueue().
createQueue will fail if the queue already exists while deployQueue will do nothing.

- Listing and closing remote connections

Client's remote addresses can be retrieved using listRemoteAddresses(). It is also possible to close the connections associated with a remote address using the closeConnectionsForAddress() method.

Alternatively, connection IDs can be listed using listConnectionIDs() and all the sessions for a given connection ID can be listed using listSessions().

- Transaction heuristic operations

In case of a server crash, when the server restarts, it possible that some transaction requires manual intervention. The listPreparedTransactions() method lists the transactions which are in the prepared states (the transactions are represented as opaque Base64 Strings.) To commit or rollback a given prepared transaction, the commitPreparedTransaction() or rollbackPreparedTransaction() method can be used to resolve heuristic transactions. Heuristically completed transactions can be listed using the listHeuristicCommittedTransactions() and listHeuristicRolledBackTransactions methods.

- Enabling and resetting Message counters

Message counters can be enabled or disabled using the enableMessageCounters() or disableMessageCounters() method. To reset message counters, it is possible to invoke resetAllMessageCounters() and resetAllMessageCounterHistories() methods.

- Retrieving the server configuration and attributes

The ActiveMQServerControl exposes Apache ActiveMQ Artemis server configuration through all its attributes (e.g. getVersion() method to retrieve the server's version, etc.)

- Listing, creating and destroying Core bridges and diverts

A list of deployed core bridges (resp. diverts) can be retrieved using the getBridgeNames() (resp. getDivertNames()) method.

Core bridges (resp. diverts) can be created or destroyed using the management operations createBridge() and destroyBridge() (resp. createDivert() and destroyDivert()).

Diverts can be updated using the management operation updateDivert().

- It is possible to stop the server and force failover to occur with any currently attached clients.

To do this use the forceFailover() operation.

Since this method actually stops the server you will probably receive some sort of error depending on which management service you use to call it.
28.1.2. Address Management

Individual addresses can be managed using the AddressControl interface.

- Modifying roles and permissions for an address

  You can add or remove roles associated to a queue using the addRole() or removeRole() methods. You can list all the roles associated to the queue with the getRoles() method.

- Pausing and resuming an Address

  The AddressControl can pause and resume an address and all the queues that are bound to it. Newly added queue will be paused too until the address is resumed. Thus all messages sent to the address will be received but not delivered. When it is resumed, delivering will occur again.

- Blocking and un blocking an Address

  The AddressControl can block and unblock an address. A blocked address will not issue any more credit to existing producers. New producers will not be granted any credit. When the address is unblocked, credit granting will resume. In this way, it is possible to drain all the queues associated with an address to quiesce a broker in a managed way.

28.1.3. Queue Management

The bulk of the management API deals with queues. The QueueControl interface defines the queue management operations.

Most of the management operations on queues take either a single message ID (e.g. to remove a single message) or a filter (e.g. to expire all messages with a given property.)

Passing null or an empty string in the filter parameter means that the management operation will be performed on all messages in a queue.

- Expiring, sending to a dead letter address and moving messages

  Messages can be expired from a queue by using the expireMessages() method. If an expiry address is defined, messages will be sent to it, otherwise they are discarded.

  Messages can also be sent to a dead letter address with the sendMessageToDeadLetterAddress() method. It returns the number of messages which are sent to the dead letter address. If a dead letter address is not defined, message are removed from the queue and discarded.

  Messages can also be moved from a queue to another queue by using the moveMessages() method.

- Listing and removing messages

  Messages can be listed from a queue by using the listMessages() method which returns an array of Map, one Map for each message.

  Messages can also be removed from the queue by using the removeMessages() method which
returns a boolean for the single message ID variant or the number of removed messages for the filter variant. The removeMessages() method takes a filter argument to remove only filtered messages. Setting the filter to an empty string will in effect remove all messages.

• Counting messages

The number of messages in a queue is returned by the getMessageCount() method. Alternatively, the countMessages() will return the number of messages in the queue which match a given filter.

• Changing message priority

The message priority can be changed by using the changeMessagesPriority() method which returns a boolean for the single message ID variant or the number of updated messages for the filter variant.

• Message counters

Message counters can be listed for a queue with the listMessageCounter() and listMessageCounterHistory() methods (see Message Counters section). The message counters can also be reset for a single queue using the resetMessageCounter() method.

• Retrieving the queue attributes

The QueueControl exposes queue settings through its attributes (e.g. getFilter() to retrieve the queue’s filter if it was created with one, isDurable() to know whether the queue is durable or not, etc.)

• Pausing and resuming Queues

The QueueControl can pause and resume the underlying queue. When a queue is paused, it will receive messages but will not deliver them. When it’s resumed, it’ll begin delivering the queued messages, if any.

• Disabling and Enabling Queues

The QueueControl can disable and enable the underlying queue. When a queue is disabled, it will not longer have messages routed to it. When it’s enabled, it’ll begin having messages routed to it again.

This is useful where you may need to disable message routing to a queue but wish to keep consumers active to investigate issues, without causing further message build up in the queue.

28.1.4. Other Resources Management

Apache ActiveMQ Artemis allows to start and stop its remote resources (acceptors, diverts, bridges, etc.) so that a server can be taken off line for a given period of time without stopping it completely (e.g. if other management operations must be performed such as resolving heuristic transactions). These resources are:

• Acceptors
They can be started or stopped using the `start()` or `stop()` method on the `AcceptorControl` interface. The acceptors parameters can be retrieved using the `AcceptorControl` attributes (see Understanding Acceptors)

- Diverts

They can be started or stopped using the `start()` or `stop()` method on the `DivertControl` interface. Diverts parameters can be retrieved using the `DivertControl` attributes (see Diverting and Splitting Message Flows)

- Bridges

They can be started or stopped using the `start()` (resp. `stop()`) method on the `BridgeControl` interface. Bridges parameters can be retrieved using the `BridgeControl` attributes (see Core bridges)

- Broadcast groups

They can be started or stopped using the `start()` or `stop()` method on the `BroadcastGroupControl` interface. Broadcast groups parameters can be retrieved using the `BroadcastGroupControl` attributes (see Clusters)

- Cluster connections

They can be started or stopped using the `start()` or `stop()` method on the `ClusterConnectionControl` interface. Cluster connections parameters can be retrieved using the `ClusterConnectionControl` attributes (see Clusters)

28.2. Management Via JMX

Apache ActiveMQ Artemis can be managed using JMX.

The management API is exposed by Apache ActiveMQ Artemis using MBeans. By default, Apache ActiveMQ Artemis registers its resources with the domain `org.apache.activemq.artemis`. For example, the `ObjectName` to manage the anycast queue `exampleQueue` on the address `exampleAddress` is:

```java
org.apache.activemq.artemis:broker=<brokerName>,component=addresses,address="exampleAddress",subcomponent=queues,routing-type="anycast",queue="exampleQueue"
```

and the MBean is:

```java
org.apache.activemq.artemis.api.core.management.QueueControl
```

The MBean's `ObjectName` is built using the helper class `org.apache.activemq.artemis.api.core.management.ObjectNameBuilder`. Example usage of the `ObjectNameBuilder` to obtain `ActiveMQServerControl`'s name:
Managing Apache ActiveMQ Artemis using JMX is identical to management of any Java Applications using JMX. It can be done by reflection or by creating proxies of the MBeans.

### 28.2.1. Configuring JMX

By default, JMX is enabled to manage Apache ActiveMQ Artemis. It can be disabled by setting `jmx-management-enabled` to `false` in broker.xml:

```xml
<jmx-management-enabled>false</jmx-management-enabled>
```

### Role Based Authorisation for JMX

Artemis uses the Java Virtual Machine's Platform MBeanServer by default. This is guarded using role based authorisation that leverages the broker's JAAS plugin support.

The RBAC used to restrict access to Mbeans and their operations can be configured in one of two ways. Via security-settings in broker.xml, described in JMX authorization in broker.xml, or via the `authorization` element in the management.xml that is described below.

#### JMX authorisation in management.xml

There are 3 elements within the `authorization` element, `allowlist`, `default-access` and `role-access`. Let's discuss each in turn.

Allowlist contains a list of MBeans that will bypass the authorisation, this is typically used for any MBeans that are needed by the console to run etc. The default configuration is:

```xml
<allowlist>
  <entry domain="hawtio"/>
</allowlist>
```

This means that any MBean with the domain `hawtio` will be allowed access without authorisation. for instance `hawtio:plugin=artemis`. You can also use wildcards for the MBean properties so the following would also match.

```xml
<allowlist>
  <entry domain="hawtio" key="type=*"/>  
</allowlist>
```

The `allowlist` element has replaced the `whitelist` element which is now
The **role-access** defines how roles are mapped to particular MBeans and its attributes and methods. The default configuration looks like:

```xml
<role-access>
  <match domain="org.apache.activemq.artemis">
    <access method="list*" roles="view,update,amq"/>
    <access method="get*" roles="view,update,amq"/>
    <access method="is*" roles="view,update,amq"/>
    <access method="set*" roles="update,amq"/>
    <access method="*" roles="amq"/>
  </match>
</role-access>
```

This contains 1 match and will be applied to any MBean that has the domain `org.apache.activemq.artemis`. Any access to any MBeans that have this domain are controlled by the `access` elements which contain a method and a set of roles. The method being invoked will be used to pick the closest matching method and the roles for this will be applied for access. For instance if you try to invoke a method called `listMessages` on an MBean with the `org.apache.activemq.artemis` domain then this would match the `access` with the method of `list*`. You could also explicitly configure this by using the full method name, like so:

```xml
<access method="listMessages" roles="view,update,amq"/>
```

You can also match specific MBeans within a domain by adding a key attribute that is used to match one of the properties on the MBean, like:

```xml
<match domain="org.apache.activemq.artemis" key="subcomponent=queues">
  <access method="list*" roles="view,update,amq"/>
  <access method="get*" roles="view,update,amq"/>
  <access method="is*" roles="view,update,amq"/>
  <access method="set*" roles="update,amq"/>
  <access method="*" roles="amq"/>
</match>
```

You could also match a specific queue for instance:

```xml
org.apache.activemq.artemis:broker=<brokerName>,component=addresses,address="exampleAddress",subcomponent=queues,routing-type="anycast",queue="exampleQueue"
```

by configuring:

```xml
<match domain="org.apache.activemq.artemis" key="queue=exampleQueue">
  <access method="list*" roles="view,update,amq"/>
</match>
```
You can also use wildcards for the MBean properties so the following would also match, allowing prefix match for the MBean properties.

```xml
<match domain="org.apache.activemq.artemis" key="queue=example*">
  <access method="list*" roles="view,update,amq"/>
  <access method="get*" roles="view,update,amq"/>
  <access method="is*" roles="view,update,amq"/>
  <access method="set*" roles="update,amq"/>
  <access method="*" roles="amq"/>
</match>
```

In case of multiple matches, the exact matches have higher priority than the wildcard matches and the longer wildcard matches have higher priority than the shorter wildcard matches.

Access to JMX MBean attributes are converted to method calls so these are controlled via the set*, get* and is*. The * access is the catch-all for everything other method that isn't specifically matched.

The default-access element is basically the catch-all for every method call that isn’t handled via the role-access configuration. This has the same semantics as a match element.

**JMX authorization in broker.xml**

The existing security-settings in broker.xml can be used for JMX RBAC.

Using the view and edit permissions on matches in security-settings provides an alternative to the authorization section in management.xml. Using a single security model based on addresses, with reloadable configuration, simplifies operation.

An MBeanServer interceptor that delegates to the broker security manager must be configured with a JVM system property that allows it to be added to all MBeanServers in the JVM.

This is configured via a system property as follows:

```java
-Djavax.management.builder.initial
=org.apache.activemq.artemis.core.server.management.ArtemisRbacMBeanServerBuilder
```

When this property is provided, the authorization section of management.xml **MUST** be omitted/removed as that depends on an alternative MBeanServer interceptor and builder.
The security-settings match addresses used for JMX RBAC use the \textit{mops}. (shorthand for management operations) \textit{prefix}.

The MBeanServer guard maps JMX MBean ObjectNames to a hierarchical address of the general form:

\[
\text{mops.<jmx domain>.<type>.<component>.<name>[.<operation>]}
\]

for the broker domain, the domain is omitted.

For example, to give the \textit{admin} role \textit{view} and \textit{edit} permissions on all MBeans, use the following security-setting:

\[
\begin{align*}
\text{<security-setting match="mops.#">} \\
\text{<permission type="view" roles="admin"/>} \\
\text{<permission type="edit" roles="admin"/>} \\
\text{</security-setting>}
\end{align*}
\]

To grant the \textit{managerRole} role \textit{view} permission to just the \textit{activemq.management} address, target the address component with name \textit{activemq.management} and with \textit{.*} to include all operations.

\[
\begin{align*}
\text{<security-setting match="mops.address.activemq.management.*">} \\
\text{<permission type="view" roles="managerRole"/>} \\
\text{</security-setting>}
\end{align*}
\]

To ensure no user has permission to \textit{force a failover} using the broker (server control) MBean, use the following that defines the empty roles set for a particular mutating operation on the \textit{broker} component:

\[
\begin{align*}
\text{<security-setting match="mops.broker.forceFailover">} \\
\text{<permission type="edit" roles=""/>} \\
\text{</security-setting>}
\end{align*}
\]

\textbf{Local JMX Access with JConsole}

Due to the authorisation which is enabled by default Apache ActiveMQ Artemis can \textit{not} be managed locally using JConsole when connecting as a \textit{local process}. This is because JConsole does not pass any authentication information when connecting this way which means the user cannot therefore be authorised for any management operations. In order to use JConsole the user will either have to disable authorisation by completely removing the \textit{authorisation} element from \textit{management.xml} or by enabling remote access and providing the proper username and password credentials (discussed next).
Remote JMX Access

By default, remote JMX access to Artemis is disabled for security reasons.

Artemis has a JMX agent which allows access to JMX MBeans remotely. This is configured via the `connector` element in the `management.xml` configuration file. To enable this you simply add the following xml:

```
<connector connector-port="1099"/>
```

This exposes the agent remotely on the port 1099. If you were connecting via JConsole you would connect as a remote process using the service url:

```
service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
```

You’d be able to supply an appropriate user name and password in this case as well.

You can also configure the connector using the following:

- **connector-host**
  The host to expose the agent on.

- **connector-port**
  The port to expose the agent on.

- **rmi-registry-port**
  The port that the RMI registry binds to. If not set, the port is always random. Set to avoid problems with remote JMX connections tunnelled through firewall.

- **jmx-realm**
  The jmx realm to use for authentication, defaults to `activemq` to match the JAAS configuration.

- **object-name**
  The object name to expose the remote connector on; default is `connector:name=rmi`.

- **secured**
  Whether the connector is secured using SSL.

- **key-store-path**
  The location of the keystore.

- **key-store-password**
  The keystore password. This can be masked.

- **key-store-provider**
  The provider; `JKS` by default.
**trust-store-path**
The location of the truststore.

**trust-store-password**
The trustore password. This can be masked.

**trust-store-provider**
The provider; JKS by default.

**password-codec**
The fully qualified class name of the password codec to use. See the password masking documentation for more details on how this works.

It is important to note that the rmi registry will pick an ip address to bind to, If you have a multi IP addresses/NICs present on the system then you can choose the ip address to use by adding the following to artemis.profile

```
-Djava.rmi.server.hostname=localhost
```

Remote connections using the default JVM Agent not enabled by default as Artemis exposes the MBean Server via its own configuration. This is so Artemis can leverage the JAAS authentication layer via JMX. If you want to expose this then you will need to disable both the connector and the authorisation by removing them from the management.xml configuration. Please refer to Java Management guide to configure the server for remote management (system properties must be set in artemis.profile).

By default, Apache ActiveMQ Artemis server uses the JMX domain "org.apache.activemq.artemis". To manage several Apache ActiveMQ Artemis servers from the same MBeanServer, the JMX domain can be configured for each individual Apache ActiveMQ Artemis server by setting jmx-domain in broker.xml:

```
<!-- use a specific JMX domain for ActiveMQ Artemis MBeans -->
<jmx-domain>my.org.apache.activemq</jmx-domain>
```

**28.2.2. Example**

See the JMX Management Example which shows how to use a remote connection to JMX and MBean proxies to manage Apache ActiveMQ Artemis.

**28.2.3. Exposing JMX using Jolokia**

The default Broker configuration ships with the Jolokia HTTP agent deployed as a web application. Jolokia is a remote JMX-over-HTTP bridge that exposes MBeans. For a full guide as to how to use it refer to Jolokia Documentation, however a simple example to query the broker's version would be to use a curl command like this:
$ curl -v -H "Origin: http://localhost" -u myUser:myPass
http://localhost:8161/console/jolokia/read/org.apache.activemq.artemis:broker="0.0.0.0"/Active

Or you could send a JSON formatted request instead of using the URL

$ curl -v -H "Origin: http://localhost" -u myUser:myPass --header "Content-type: application/json" --request POST --data '{"attribute": "Active", "mbean": "org.apache.activemq.artemis:broker="0.0.0.0"", "type": "read"}'
http://localhost:8161/console/jolokia

By default it's necessary to pass the Origin header due to the CORS checking which is configured in etc/jolokia-access.xml.

Such a curl command would give you back something like the following (after formatting):

```json
{
  "request": {
    "mbean": "org.apache.activemq.artemis:broker="0.0.0.0"",
    "attribute": "Version",
    "type": "read"
  },
  "value": "2.24.0",
  "timestamp": 1663086398,
  "status": 200
}
```

### 28.2.4. JMX and the Web Console

The web console that ships with Artemis uses Jolokia under the covers which in turn uses JMX. This will use the authentication configuration as described in the Role Based Authorisation for JMX section. This means that when MBeans are accessed via the console the credentials used to log into the console and the roles associated with them. By default, access to the console is only allow via users with the amq role. This is configured in the artemis.profile via the system property -Dhawtio.role=amq. You can configure multiple roles by changing this to -Dhawtio.roles=amq:view:update.

If a user doesn't have the correct role to invoke a specific operation then this will display an authorisation exception in the console.

**ArtemisRbacMBeanServerBuilder and ArtemisRbacInvocationHandler**

The ArtemisRbacMBeanServerBuilder class, when configured as value for the system property javax.management.builder.initial will cause the ArtemisRbacInvocationHandler to be installed on every JMX MBeanServer in the JVM. The ArtemisRbacInvocationHandler intercepts all operations on the MBeanServer and chooses to guard a subsection of those operations.
For guarded operations the \textit{view} or \textit{edit} permissions are required to make an invocation. If the current authenticated subject does not have the required roles to grant those permissions, a security exception is thrown.

For query operations on the MBeanServer, the results of the query are limited to entries that have the required \textit{view} permission.

\section*{28.3. Using Management Message API}

The management message API in ActiveMQ Artemis is accessed by sending Core Client messages to a special address, the \textit{management address}.

\textit{Management messages} are regular Core Client messages with well-known properties that the server needs to understand to interact with the management API:

- The name of the managed resource
- The name of the management operation
- The parameters of the management operation

When such a management message is sent to the management address, Apache ActiveMQ Artemis server will handle it, extract the information, invoke the operation on the managed resources and send a \textit{management reply} to the management message's reply-to address (specified by \texttt{ClientMessageImpl.REPLYTO_HEADER_NAME}).

A \texttt{ClientConsumer} can be used to consume the management reply and retrieve the result of the operation (if any) stored in the reply's body. For portability, results are returned as a \texttt{JSON} String rather than Java Serialization (the \texttt{org.apache.activemq.artemis.api.core.management.ManagementHelper} can be used to convert the JSON string to Java objects).

These steps can be simplified to make it easier to invoke management operations using Core messages:

1. Create a \texttt{ClientRequestor} to send messages to the management address and receive replies
2. Create a \texttt{ClientMessage}
3. Use the helper class \texttt{org.apache.activemq.artemis.api.core.management.ManagementHelper} to fill the message with the management properties
4. Send the message using the \texttt{ClientRequestor}
5. Use the helper class \texttt{org.apache.activemq.artemis.api.core.management.ManagementHelper} to retrieve the operation result from the management reply.

For example, to find out the number of messages in the queue \texttt{exampleQueue}:

\begin{verbatim}
ClientSession session = ...;
ClientRequestor requestor = new ClientRequestor(session, "activemq.management");
ClientMessage message = session.createMessage(false);
ManagementHelper.putAttribute(message, "queue.exampleQueue", "messageCount");
\end{verbatim}
Management operation name and parameters must conform to the Java interfaces defined in the management packages.

Names of the resources are built using the helper class `org.apache.activemq.artemis.api.core.management.ResourceNames` and are straightforward (e.g. `queue.exampleQueue` for `QueueControl` of the Queue `exampleQueue`, or `broker` for the `ActiveMQServerControl`).

The `ManagementHelper` class can be used only with Core JMS messages. When called with a message from a different JMS library, an exception will be thrown.

### 28.3.1. Configuring Management

The management address to send management messages is configured in `broker.xml`:

```
<management-address>activemq.management</management-address>
```

By default, the address is `activemq.management`.

The management address requires a special user permission `manage` to be able to receive and handle management messages. This is also configured in `broker.xml`:

```
<!-- users with the admin role will be allowed to manage -->
<!-- Apache ActiveMQ Artemis using management messages -->
<security-setting match="activemq.management">
  <permission type="manage" roles="admin" />
</security-setting>
```

### 28.3.2. Fine grained RBAC on management messages

There is optional RBAC on the content of management messages sent to the management address. RBAC is enabled through configuration by setting the attribute `management-message-rbac` to `true`.

The `manage` permission is required to execute management operations via messages. The `view` and `edit` permissions must be used in conjunction with the `manage` permission.

When enabled, more fine-grained permissions on the content of management messages sent to the management address can be configured through the security-settings.

The security-settings match addresses used for RBAC follow the general hierarchical form of:
management-rbac-prefix, component type, component name, operation. Where the values are extracted from the management message headers.

<management-rbac-prefix>.<resource type>.<resource name>.<operation>

Immutable operations and attribute access will require the view permission, all other operations will require the edit permission.

In the following example the dataImport role can only access the id attribute of queues, which is the only management operation that is required by the data import command line tool.

<security-setting match="mops.queue.*.getID">
    <permission type="view" roles="dataImport" />
    <permission type="manage" roles="dataImport" />
</security-setting>

If you want the admin role to have full access, use a wildcard after the management-rbac-prefix and grant both the view and edit permissions:

<security-setting match="mops.#">
    <permission type="view" roles="admin" />
    <permission type="update" roles="admin" />
    <permission type="manage" roles="admin" />
</security-setting>

28.3.3. Management Example

See the Management Example which shows how to use JMS messages to manage the Apache ActiveMQ Artemis server.

28.4. Management Notifications

Apache ActiveMQ Artemis emits notifications to inform listeners of potentially interesting events (creation of new resources, security violation, etc.).

These notifications can be received by two different ways:

- JMX notifications
- Notification messages

28.4.1. JMX Notifications

If JMX is enabled (see Configuring JMX section), JMX notifications can be received by subscribing to org.apache.activemq.artemis:type=Broker,brokerName=<broker name>,module=Core,serviceType=Server for notifications on resources.
28.4.2. Notification Messages

Apache ActiveMQ Artemis defines a special management notification address. Queues can be bound to this address so that clients will receive management notifications as messages.

A client which wants to receive management notifications must create a queue bound to the management notification address. It can then receive the notifications from its queue.

Notifications messages are regular messages with additional properties corresponding to the notification (its type, when it occurred, the resources which were concerned, etc.).

Since notifications are regular messages, it is possible to use message selectors to filter out notifications and receives only a subset of all the notifications emitted by the server.

Configuring The Management Notification Address

The management notification address to receive management notifications is configured in broker.xml:

```xml
<management-notification-address>activemq.notifications</management-notification-address>
```

By default, the address is activemq.notifications.

Suppressing Session Notifications

Some messaging patterns can generate a lot of SESSION CREATED and SESSION CLOSED notifications. In a clustered environment this will come with some computational overhead. If these notifications are not otherwise used they can be disabled through:

```xml
<suppress-session-notifications>true</suppress-session-notifications>
```

The only time these notifications are required is in a cluster with MQTT clients where unique client ID utilization needs to be enforced. Default value is false.

Receiving Notification Messages

Apache ActiveMQ Artemis’s Core JMS Client can be used to receive notifications:

```java
Topic notificationsTopic = ActiveMQJMSClient.createTopic("activemq.notifications");

Session session = ...
MessageConsumer notificationConsumer = session.createConsumer(notificationsTopic);
notificationConsumer.setMessageListener(new MessageListener() {
    public void onMessage(Messagenotif) {
        System.out.println("------------------------");
        System.out.println("Received notification:");
        try {
            
```
28.4.3. Example

See the Management Notification Example which shows how to use a JMS MessageListener to receive management notifications from ActiveMQ Artemis server.

28.4.4. Notification Types and Headers

Below is a list of all the different kinds of notifications as well as which headers are on the messages. Every notification has a _AMQ_NotifType (value noted in parentheses) and _AMQ_NotifTimestamp header. The timestamp is the un-formatted result of a call to java.lang.System.currentTimeMillis().

**BINDING_ADDED (0)**

_AMQ_Binding_Type, _AMQ_Address, _AMQ_ClusterName, _AMQ_RoutingName, _AMQ_Binding_ID, _AMQ_Distance, _AMQ_FilterString

**BINDING_REMOVED (1)**

_AMQ_Address, _AMQ_ClusterName, _AMQ_RoutingName, _AMQ_Binding_ID, _AMQ_Distance, _AMQ_FilterString

**CONSUMER_CREATED (2)**

_AMQ_Address, _AMQ_ClusterName, _AMQ_RoutingName, _AMQ_Distance, _AMQ_ConsumerCount, _AMQ_User, _AMQ_ValidatedUser, _AMQ_RemoteAddress, _AMQ_SessionName, _AMQ_FilterString, _AMQ_CertSubjectDN

**CONSUMER_CLOSED (3)**

_AMQ_Address, _AMQ_ClusterName, _AMQ_RoutingName, _AMQ_Distance, _AMQ_ConsumerCount, _AMQ_User, _AMQ_RemoteAddress, _AMQ_SessionName, _AMQ_FilterString

**SECURITY_AUTHENTICATION_VIOLATION (6)**

_AMQ_User, _AMQ_CertSubjectDN, _AMQ_RemoteAddress

**SECURITY_PERMISSION_VIOLATION (7)**

_AMQ_Address, _AMQ_CheckType, _AMQ_User

**DISCOVERY_GROUP_STARTED (8)**

name
DISCOVERY_GROUP_STOPPED (9)
  name

BROADCAST_GROUP_STARTED (10)
  name

BROADCAST_GROUP_STOPPED (11)
  name

BRIDGE_STARTED (12)
  name

BRIDGE_STOPPED (13)
  name

CLUSTER_CONNECTION_STARTED (14)
  name

CLUSTER_CONNECTION_STOPPED (15)
  name

ACCEPTOR_STARTED (16)
  factory, id

ACCEPTOR_STOPPED (17)
  factory, id

PROPOSAL (18)
  _JBM_ProposalGroupId, _JBM_ProposalValue, _AMQ_Binding_Type, _AMQ_Address, _AMQ_Distance

PROPOSAL_RESPONSE (19)
  _JBM_ProposalGroupId, _JBM_ProposalValue, _JBM_ProposalAltValue, _AMQ_Binding_Type, _AMQ_Address, _AMQ_Distance

CONSUMER_SLOW (21)
  _AMQ_Address, _AMQ_ConsumerCount, _AMQ_RemoteAddress, _AMQ_ConnectionName, _AMQ_ConsumerName, _AMQ_SessionName

ADDRESS_ADDED (22)
  _AMQ_Address, _AMQ_Routing_Type

ADDRESS_REMOVED (23)
  _AMQ_Address, _AMQ_Routing_Type

CONNECTION_CREATED (24)
  _AMQ_ConnectionName, _AMQ_RemoteAddress
CONNECTION_DESTROYED (25)
   _AMQ_ConnectionName, _AMQ_RemoteAddress

SESSION_CREATED (26)
   _AMQ_ConnectionName, _AMQ_User, _AMQ_SessionName

SESSION_CLOSED (27)
   _AMQ_ConnectionName, _AMQ_User, _AMQ_SessionName

MESSAGE_DELIVERED (28)
   _AMQ_Address, _AMQ_Routing_Type, _AMQ_RoutingName, _AMQ_ConsumerName, _AMQ_Message_ID

MESSAGE_EXPIRED (29)
   _AMQ_Address, _AMQ_Routing_Type, _AMQ_RoutingName, _AMQ_ConsumerName, _AMQ_Message_ID

28.5. Message Counters

Message counters can be used to obtain information on queues over time as Apache ActiveMQ Artemis keeps a history on queue metrics.

They can be used to show trends on queues. For example, using the management API, it would be possible to query the number of messages in a queue at regular interval. However, this would not be enough to know if the queue is used: the number of messages can remain constant because nobody is sending or receiving messages from the queue or because there are as many messages sent to the queue than messages consumed from it. The number of messages in the queue remains the same in both cases but its use is widely different.

Message counters give additional information about the queues:

**count**
   The total number of messages added to the queue since the server was started

**countDelta**
   the number of messages added to the queue since the last message counter update

**messageCount**
   The current number of messages in the queue

**messageCountDelta**
   The overall number of messages added/removed from the queue since the last message counter update. For example, if messageCountDelta is equal to -10 this means that overall 10 messages have been removed from the queue (e.g. 2 messages were added and 12 were removed)

**lastAddTimestamp**
   The timestamp of the last time a message was added to the queue

**lastAckTimestamp**
   The timestamp of the last time a message from the queue was acknowledged
updateTimestamp

The timestamp of the last message counter update

These attributes can be used to determine other meaningful data as well. For example, to know specifically how many messages were consumed from the queue since the last update simply subtract the messageCountDelta from countDelta.

28.5.1. Configuring Message Counters

By default, message counters are disabled as it might have a small negative effect on memory.

To enable message counters, you can set it to true in broker.xml:

```xml
<message-counter-enabled>true</message-counter-enabled>
```

Message counters keep a history of the queue metrics (10 days by default) and sample all the queues at regular interval (10 seconds by default). If message counters are enabled, these values should be configured to suit your messaging use case in broker.xml:

```xml
<!-- keep history for a week -->
<message-counter-max-day-history>7</message-counter-max-day-history>
<!-- sample the queues every minute (60000ms) -->
<message-counter-sample-period>60000</message-counter-sample-period>
```

Message counters can be retrieved using the Management API. For example, to retrieve message counters on a queue using JMX:

```java
// retrieve a connection to Apache ActiveMQ Artemis's MBeanServer
MBeanServer mbsc = ...
QueueControlMBean queueControl = (QueueControlMBean) MBeanServerInvocationHandler.
.newProxyInstance(mbsc,
    on,
    QueueControl.class,
    false);
// message counters are retrieved as a JSON String
String counters = queueControl.listMessageCounter();
// use the MessageCounterInfo helper class to manipulate message counters more easily
MessageCounterInfo messageCounter = MessageCounterInfo.fromJSON(counters);
System.out.format("%s message(s) in the queue (since last sample: %s)\n",
    messageCounter.getMessageCount(),
    messageCounter.getMessageCountDelta());
```

28.5.2. Example

See the Message Counter Example which shows how to use message counters to retrieve information on a queue.
Chapter 29. Management Console

Apache ActiveMQ Artemis ships by default with a management console. It is powered by Hawt.io.

Its purpose is to expose the Management API via a user friendly web ui.

29.1. Login

To access the management console use a browser and go to the URL http://localhost:8161/console.

A login screen will be presented, if your broker is secure, you will need to use a user with admin role, if it is unsecure simply enter any user/password.

![Login Screen](image)

29.2. Security

That Jolokia JMX-HTTP bridge is secured via a policy file in the broker configuration directory: 'etc/jolokia-access.xml'. The contents of that file should be modified as described in the Jolokia Security Guide. By default the console is locked down to 'localhost', pay particular attention to the 'CORS' restrictions when exposing the console web endpoint over the network.

29.3. Console

Once logged in you should be presented with a screen similar to.
29.3.1. Navigation Menu

On the top right is small menu area you will see some icons.

- **question mark** This will open a menu with the following items
- **Help** This will navigate to the console user guide
- **About** this will load an about screen, here you will be able to see and validate versions
- **person** will provide a drop down menu with
- **Preferences** this will open the preferences page
- **Log out** self descriptive.

29.3.2. Navigation Tabs

Running below the Navigation Menu you will see several default feature tabs.

- **Artemis** This is the core tab for Apache ActiveMQ Artemis specific functionality. The rest of this document will focus on this.
- **Dashboard** Here you can create and save graphs and tables of metrics available via JMX, a default jvm health dashboard is provided.
- **JMX** This exposes the raw Jolokia JMX so you can browse/access all the JMX endpoints exposed by the JVM.
- **Threads** This allows you to monitor the thread usage and their state.

In previous versions there was a "Connect" tab which could be used to connect to a remote broker from the same console. This was disabled by default for security purposes, but it can be enabled again by removing `-Dhawtio.disableProxy=true` from `artemis.profile` (or `artemis.profile.cmd` on Windows).
You can install further hawtio plugins if you wish to have further functionality.

29.4. Artemis Tab

Click Artemis in the left navigation bar to see the Artemis specific plugin. (The Artemis tab won’t appear if there is no broker in this JVM). The Artemis plugin works very much the same as the JMX plugin however with a focus on interacting with an Artemis broker.

29.4.1. Tree View

The tree view on the left-hand side shows the top level JMX tree of each broker instance running in the JVM. Expanding the tree will show the various MBeans registered by Artemis that you can inspect via the Attributes tab.

Acceptors

This expands to show and expose details of the current configured acceptors.

Addresses

This expands to show the current configured available addresses.

Under the address you can expand to find the queues for the address exposing attributes

29.4.2. Key Operations

Creating a new Address

To create a new address simply click on the broker or the address folder in the jmx tree and click on the create tab.

Once you have created an address you should be able to Send to it by clicking on it in the jmx tree and clicking on the send tab.

Creating a new Queue

To create a new queue click on the address you want to bind the queue to and click on the create tab.

Once you have created a queue you should be able to Send a message to it or Browse it or view the Attributes or Charts. Simply click on the queue in the jmx tree and click on the appropriate tab.

You can also see a graphical view of all brokers, addresses, queues and their consumers using the Diagram tab.

29.4.3. Status Logging

When the broker starts it will detect the presence of the web console and log status information, e.g.:
The web console is detected by inspecting the value of the `<display-name>` tag in the war file’s `WEB-INF/web.xml` descriptor. By default it looks for `hawtio`. However, if this value is changed for any reason the broker can look for this new value by setting the following system property

-Dorg.apache.activemq.artemis.webConsoleDisplayName=newValue
Chapter 30. Metrics

Apache ActiveMQ Artemis can export metrics to a variety of monitoring systems via the Micrometer vendor-neutral application metrics facade.

Important runtime metrics have been instrumented via the Micrometer API, and all a user needs to do is implement org.apache.activemq.artemis.core.server.metrics.ActiveMQMetricsPlugin in order to instantiate and configure a io.micrometer.core.instrument.MeterRegistry implementation. Relevant implementations of MeterRegistry are available from the Micrometer code-base.

This is a simple interface:

```java
public interface ActiveMQMetricsPlugin extends Serializable {
    ActiveMQMetricsPlugin init(Map<String, String> options);
    MeterRegistry getRegistry();
    default void registered(ActiveMQServer server) { }
}
```

When the broker starts it will call init and pass in the options which can be specified in XML as key/value properties. At this point the plugin should instantiate and configure the io.micrometer.core.instrument.MeterRegistry implementation.

Later during the broker startup process it will call getRegistry in order to get the MeterRegistry implementation and use it for registering meters. Once registered, it will call registered to provide the plugin with a handle to the server. The plugin can then use that handle later to inspect whether the broker is operational and not in a startup or shutdown phase.

The broker ships with two ActiveMQMetricsPlugin implementations:

org.apache.activemq.artemis.core.server.metrics.plugins.LoggingMetricsPlugin
- This plugin simply logs metrics. It’s not very useful for production, but can serve as a demonstration of the Micrometer integration. It takes no key/value properties for configuration.

org.apache.activemq.artemis.core.server.metrics.plugins.SimpleMetricsPlugin
- This plugin is used for testing. It is in-memory only and provides no external output. It takes no key/value properties for configuration.

30.1. Exported Metrics

The following metrics are exported, categorized by component. A description for each metric is exported along with the metric itself therefore the description will not be repeated here.

Every metric is “tagged” with the broker tag (configured via <name> in broker.xml). A tag is a piece of metadata that gives context to the metric. These tags are the foundation of what is sometimes referred to as “dimensional metrics.” Metrics may have additional tags, but at the very least they
will all have the broker tag.

Lastly, all metrics specifically for ActiveMQ Artemis are prefixed with artemis.

### 30.1.1. Broker

- connection.count
- total.connection.count
- address.memory.usage
- address.memory.usage.percentage
- disk.store.usage
- replica.sync
- active
- authentication.count tagged by result - either success or failure
- authorization.count tagged by result - either success or failure

### 30.1.2. Address

These metrics are tagged with the address tag which reflects the name of the corresponding address.

- routed.message.count
- unrouted.message.count
- address.size
- number.of.pages

### 30.1.3. Queue

These metrics are tagged with the address & queue tags which reflects the name of the corresponding address & queue respectively.

- message.count
- durable.message.count
- persistent.size
- durable.persistent.size
- delivering.message.count
- delivering.durable.message.count
- delivering.persistent.size
- delivering.durable.persistent.size
- scheduled.message.count
- scheduled.durable.message.count
It may appear that some higher level broker metrics are missing (e.g. total message count). However, these metrics can be deduced by aggregating the lower level metrics (e.g. aggregate the message.count metrics from all queues to get the total).

### 30.1.4. Optional metrics

There are a handful of other useful metrics that are related to the JVM, the underlying operating system, etc. These metrics are provided specifically by Micrometer and therefore do not have the artemis. prefix.

**JVM memory metrics**

Gauges buffer and memory pool utilization. Underlying data gathered from Java’s BufferPoolMXBeans and MemoryPoolMXBeans.

Enabled by default.

**JVM GC**

Gauges max and live data size, promotion and allocation rates, and the number of times the GC pauses (or concurrent phase time in the case of CMS). Underlying data gathered from Java’s MemoryPoolMXBeans.

Disabled by default.

**JVM thread**

Gauges thread peak, the number of daemon threads, and live threads. Underlying data gathered from Java’s ThreadMXBean.

Disabled by default.

**Netty**

Collects metrics from Netty’s PooledByteBufAllocatorMetric.

Disabled by default.

**File descriptors**

Gauges current and max-allowed open files.

Disabled by default.
**Processor**

Gauges system CPU count, CPU usage, and 1-minute load average as well as process CPU usage.

Disabled by default.

**Uptime**

Gauges process start time and uptime.

Disabled by default.

**Logging**

Counts the number of logging events per logging category (e.g. `WARN`, `ERROR`, etc.).

Disabled by default.

This works *exclusively* with Log4j2 (i.e the default logging implementation shipped with the broker). If you're embedding the broker and using a different logging implementation (e.g. Log4j 1.x, JUL, Logback, etc.) and you enable these metrics then the broker will fail to start with a `java.lang.NoClassDefFoundError` as it attempts to locate Log4j2 classes that don’t exist on the classpath.

**Security caches**

The following authentication & authorization cache metrics are exported. They are all tagged by `cache` (either `authentication` or `authorization`). Additional tags are noted.

- `cache.size`
- `cache.puts`
- `cache.gets` tagged by `result` - either `hit` or `miss`
- `cache.evictions`
- `cache.eviction.weight`

Disabled by default.

**30.2. Configuration**

Metrics for all addresses and queues are enabled by default. If you want to disable metrics for a particular address or set of addresses you can do so by setting the `enable-metrics address-setting` to `false`.

In `broker.xml` use the `metrics` element to configure which general broker and JVM metrics are reported and to configure the plugin itself. Here's a configuration with all optional metrics:

```xml
<metrics>
  <jvm-memory>true</jvm-memory> <!-- defaults to true -->
  <jvm-gc>true</jvm-gc> <!-- defaults to false -->
  <jvm-threads>true</jvm-threads> <!-- defaults to false -->
  <netty-pool>true</netty-pool> <!-- defaults to false -->
  <file-descriptors>true</file-descriptors> <!-- defaults to false -->
</metrics>
```
The plugin can also be configured with key/value properties in order to customize the implementation as necessary, e.g.:

```
<metrics>
  <plugin class-name="org.example.MyMetricsPlugin">
    <property key="host" value="example.org" />
    <property key="port" value="5162" />
    <property key="foo" value="10" />
  </plugin>
</metrics>
```
Chapter 31. Core Bridges

The function of a bridge is to consume messages from a source queue, and forward them to a target address, typically on a different Apache ActiveMQ Artemis server.

The source and target servers do not have to be in the same cluster which makes bridging suitable for reliably sending messages from one cluster to another, for instance across a WAN, or internet and where the connection may be unreliable.

The bridge has built in resilience to failure so if the target server connection is lost, e.g. due to network failure, the bridge will retry connecting to the target until it comes back online. When it comes back online it will resume operation as normal.

In summary, bridges are a way to reliably connect two separate Apache ActiveMQ Artemis servers together. With a core bridge both source and target servers must be Apache ActiveMQ Artemis servers.

Bridges can be configured to provide once and only once delivery guarantees even in the event of the failure of the source or the target server. They do this by using duplicate detection (described in Duplicate Detection).

Although they have similar function, don't confuse core bridges with JMS bridges!

Core bridges are for linking an Apache ActiveMQ Artemis node with another Apache ActiveMQ Artemis node and do not use the JMS API. A JMS Bridge is used for linking any two JMS 1.1 compliant JMS providers. So, a JMS Bridge could be used for bridging to or from different JMS compliant messaging system. It’s always preferable to use a core bridge if you can. Core bridges use duplicate detection to provide once and only once guarantees. To provide the same guarantee using a JMS bridge you would have to use XA which has a higher overhead and is more complex to configure.

31.1. Configuring Core Bridges

Bridges are configured in broker.xml. Let's kick off with an example:

```xml
<bridge name="my-bridge">
    <queue-name>sausage-factory</queue-name>
    <forwarding-address>mincing-machine</forwarding-address>
    <ha>true</ha>
    <filter string="name='aardvark'"/>
    <transformer-class-name>
        org.apache.activemq.artemis.jms.example.HatColourChangeTransformer
    </transformer-class-name>
    <min-large-message-size>102400</min-large-message-size>
    <check-period>30000</check-period>
    <connection-ttl>60000</connection-ttl>
    <retry-interval>2000</retry-interval>
</bridge>
```
In the above example we have shown all the parameters it's possible to configure for a bridge. In practice you might use many of the defaults so it won't be necessary to specify them all explicitly.

Let's take a look at all the parameters in turn:

**name**

All bridges must have a unique name in the server.

**queue-name**

This is the unique name of the local queue that the bridge consumes from, it's a mandatory parameter.

The queue must already exist by the time the bridge is instantiated at start-up.

**forwarding-address**

This is the address on the target server that the message will be forwarded to. If a forwarding address is not specified, then the original address of the message will be retained.

**ha**

This optional parameter determines whether or not this bridge should support high availability. True means it will connect to any available server in a cluster and support failover. The default value is false.

**filter-string**

An optional filter string can be supplied. If specified then only messages which match the filter expression specified in the filter string will be forwarded. The filter string follows the ActiveMQ Artemis filter expression syntax described in Filter Expressions.
**transformer-class-name**

An *optional* transformer can be specified. This gives you the opportunity to transform the message's header or body before forwarding it. See the [transformer chapter](#) for more details about transformer-specific configuration.

**min-large-message-size**

Any message larger than this size (in bytes) is considered a large message (to be sent in chunks). Supports byte notation like "K", "Mb", "MiB", "GB", etc. Default is 102400 (i.e. 100KiB).

**check-period**

Sets the period (in milliseconds) used to check if the bridge client has failed to receive pings from the server. Use `-1` to disable this check. Default is 30000.

**connection-ttl**

How long (in milliseconds) the remote server will keep the connection alive in the absence of any data arriving from the bridge. This should be greater than the check-period. Default is 60000.

**retry-interval**

This optional parameter determines the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is 2000 milliseconds.

**retry-interval-multiplier**

This optional parameter determines a multiplier to apply to the time since the last retry to compute the time to the next retry. This allows you to implement an *exponential backoff* between retry attempts.

Let's take an example:

If we set `retry-interval` to 1000 ms and we set `retry-interval-multiplier` to 2.0, then, if the first reconnect attempt fails, we will wait 1000 ms then 2000 ms then 4000 ms between subsequent reconnection attempts.

The default value is 1.0 meaning each reconnect attempt is spaced at equal intervals.

**max-retry-interval**

This enforces a limit on `retry-interval` since it can grow due to `retry-interval-multiplier`. Default is 2000.

**initial-connect-attempts**

This optional parameter determines the total number of initial connect attempts the bridge will make before giving up and shutting down. A value of `-1` signifies an unlimited number of attempts. The default value is `-1`.

**reconnect-attempts**

This optional parameter determines the total number of reconnect attempts the bridge will make before giving up and shutting down. A value of `-1` signifies an unlimited number of attempts. The default value is `-1`. 

---

A transformer is a function that operates on the data of a message before it is forwarded. Transformers can be specified for the bridge and are used to transform the message's header or body before forwarding it. Transformers can be useful for tasks such as encrypting data, adding metadata, or filtering messages.

The `min-large-message-size` parameter determines the size threshold for a message to be considered large. Messages larger than this size will be sent in chunks to avoid overloading the network.

`check-period` is the interval used to check if the bridge client has failed to receive pings from the server. The default value is 30 seconds, but this can be increased to make the bridge more robust against network disruptions.

`connection-ttl` is the amount of time the remote server will keep the connection alive in the absence of any data. This should be longer than the `check-period` to ensure that the bridge can still receive data after a connection failure.

`retry-interval` is the period between subsequent reconnection attempts. If the connection to the target server fails, the bridge will attempt to reconnect at intervals specified by this parameter. The default value is 2 seconds.

`retry-interval-multiplier` allows for exponential backoff, where the time between retries is multiplied by a factor. This helps to avoid overwhelming the server with too many retries.

`max-retry-interval` limits the maximum value that `retry-interval` can reach due to `retry-interval-multiplier`. This prevents the number of retries from becoming too large.

`initial-connect-attempts` and `reconnect-attempts` determine how many attempts the bridge will make before giving up and shutting down. A value of `-1` means this parameter is unlimited.

---

A large message is one that exceeds the size specified by the `min-large-message-size` parameter. Messages of this size are often sent in chunks to prevent network congestion.

The `check-period` is a safeguard against disconnection, allowing the bridge to detect when it stops receiving pings from the server.

The `connection-ttl` parameter ensures that the connection remains open even if there's no data to send, providing a buffer for reconnects.

The `retry-interval` and `retry-interval-multiplier` work together to manage the reconnection attempts, ensuring that the bridge doesn't constantly attempt to reconnect unnecessarily.

The `max-retry-interval` prevents the `retry-interval` from growing indefinitely due to the `retry-interval-multiplier`, maintaining a reasonable retry rate.

The `initial-connect-attempts` and `reconnect-attempts` parameters control the maximum number of attempts the bridge will make before giving up, providing a balance between retrying too much and giving up too soon.
use-duplicate-detection

This optional parameter determines whether the bridge will automatically insert a duplicate id property into each message that it forwards.

Doing so, allows the target server to perform duplicate detection on messages it receives from the source server. If the connection fails or server crashes, then, when the bridge resumes it will resend unacknowledged messages. This might result in duplicate messages being sent to the target server. By enabling duplicate detection allows these duplicates to be screened out and ignored.

This allows the bridge to provide a once and only once delivery guarantee without using heavyweight methods such as XA (see Duplicate Detection for more information).

The default value for this parameter is true.

confirmation-window-size

This optional parameter determines the confirmation-window-size to use for the connection used to forward messages to the target node. Supports byte notation like "K", "Mb", "MiB", "GB", etc. This attribute is described in section Client failover attributes

When using the bridge to forward messages to an address which uses the BLOCK address-full-policy from a queue which has a max-size-bytes set it's important that confirmation-window-size is less than or equal to max-size-bytes to prevent the flow of messages from ceasing.

producer-window-size

This optional parameter determines the producer flow control through the bridge. Use -1 to disable. Supports byte notation like "K", "Mb", "MiB", "GB", etc. Default is 1048576 (i.e. 1MiB).

user

This optional parameter determines the user name to use when creating the bridge connection to the remote server. If it is not specified the default cluster user specified by cluster-user in broker.xml will be used.

password

This optional parameter determines the password to use when creating the bridge connection to the remote server. If it is not specified the default cluster password specified by cluster-password in broker.xml will be used.

reconnect-attempts-same-node

This configures the number of times reconnection attempts will be made to the same node on the topology before reverting back to the initial connector(s). Default is 10.

routing-type

Bridges can apply a particular routing-type to the messages it forwards, strip the existing routing type, or simply pass the existing routing-type through. This is useful in situations where the message may have its routing-type set but you want to bridge it to an address using a different routing-type. It's important to keep in mind that a message with the anycast routing-type will not
actually be routed to queues using multicast and vice-versa. By configuring the routing-type of the bridge you have the flexibility to deal with any situation. Valid values are ANYCAST, MULTICAST, PASS, & STRIP. The default is PASS.

**concurrency**

For bridging high latency networks, and particularly for destinations with a high throughput, more workers might have to be committed to the bridge. This is done with the concurrency parameter. Increasing the concurrency will get reflected by more consumers and producers showing up on the bridged destination, allowing for increased parallelism across high latency networks. The default is 1.

When using a concurrency value greater than 1 multiple bridges will be created and named with an index. For example, if a bridge named myBridge was configured with a concurrency of 3 then actually 3 bridges would be created named myBridge-0, myBridge-1, and myBridge-2. This is important to note for management operations as each bridge will have its own associated BridgeControl.

**static-connectors**

Pick either this or discovery-group-ref to connect the bridge to the target server.

The static-connectors is a list of connector-ref elements pointing to connector elements defined elsewhere. A connector encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc) as well as the server connection parameters (host, port etc). For more information about what connectors are and how to configure them, please see Configuring the Transport.

**discovery-group-ref**

Pick either this or static-connector to connect the bridge to the target server.

The discovery-group-ref element has one attribute - discovery-group-name. This attribute points to a discovery-group defined elsewhere. For more information about what discovery-groups are and how to configure them, please see Discovery Groups.
Chapter 32. Clusters

32.1. Overview

Apache ActiveMQ Artemis clusters allow groups of Apache ActiveMQ Artemis servers to be grouped together in order to share message processing load. Each active node in the cluster is an active Apache ActiveMQ Artemis server which manages its own messages and handles its own connections.

The cluster is formed by each node declaring cluster connections to other nodes in the core configuration file broker.xml. When a node forms a cluster connection to another node, internally it creates a core bridge (as described in Core Bridges) connection between it and the other node, this is done transparently behind the scenes - you don’t have to declare an explicit bridge for each node. These cluster connections allow messages to flow between the nodes of the cluster to balance load.

Nodes can be connected together to form a cluster in many different topologies, we will discuss a couple of the more common topologies later in this chapter.

We’ll also discuss client side load balancing, where we can balance client connections across the nodes of the cluster, and we’ll consider message redistribution where Apache ActiveMQ Artemis will redistribute messages between nodes to avoid starvation.

Another important part of clustering is server discovery where servers can broadcast their connection details so clients or other servers can connect to them with the minimum of configuration.

Once a cluster node has been configured it is common to simply copy that configuration to other nodes to produce a symmetric cluster. However, care must be taken when copying the Apache ActiveMQ Artemis files. Do not copy the Apache ActiveMQ Artemis data (i.e. the bindings, journal, paging, and large-messages directories) from one node to another. When a node is started for the first time and initializes its journal files it also persists a special identifier to the journal directory. This id must be unique among nodes in the cluster or the cluster will not form properly.

32.2. Performance Considerations

It is important to note that while the goal of clustering is to increase overall message throughput via horizontal scaling it is not a “silver bullet.” In certain situations clustering can, in fact, reduce message throughput so care must be taken when choosing a clustered configuration. Here’s a few general guidelines:

1. **Establish a clear, concrete performance goal.** Performance testing & tuning are often difficult and tedious activities. Small, relative gains will tempt you to keep going, and without a goal you will never know when to stop. You need a goal to know “how good is good enough.”

2. **Start simple.** Benchmark your use-case with a single broker first. A single broker can handle millions of messages per second in certain use-cases. If you can’t meet your performance goal
with a single broker only then move to a clustered configuration. Only add complexity when there is a clear benefit.

The main way a cluster can reduce overall message throughput is if there are not enough producers & consumers on each node leading to message build-up on some nodes and consumer starvation others. The cluster has mechanisms to deal with this (i.e. message load-balancing & redistribution, which will be covered later), but you really don’t want the broker to intervene and move messages between nodes unless absolutely necessary because that adds latency.

Therefore, when thinking in performance terms the main question one must answer when choosing a clustered configuration is: Do I have enough clients so that each node in the cluster has sufficient consumers to receive all the messages produced on that node? If the answer to that question is "yes" then clustering may, in fact, improve overall message throughput for you. If the answer to that question is "no" then you're likely to get better performance from either a smaller cluster or just a single broker.

Also keep in mind that a connection router may improve performance of your cluster by grouping related consumers and producers together on the same node.

### 32.3. Server discovery

Server discovery is a mechanism by which servers can propagate their connection details to:

- Messaging clients. A messaging client wants to be able to connect to the servers of the cluster without having specific knowledge of which servers in the cluster are up at any one time.
- Other servers. Servers in a cluster want to be able to create cluster connections to each other without having prior knowledge of all the other servers in the cluster.

This information, let’s call it the Cluster Topology, is actually sent around normal Apache ActiveMQ Artemis connections to clients and to other servers over cluster connections. This being the case we need a way of establishing the initial first connection. This can be done using dynamic discovery techniques like UDP and JGroups, or by providing a list of initial connectors.

#### 32.3.1. Dynamic Discovery

Server discovery uses UDP multicast or JGroups to broadcast server connection settings.

**Broadcast Groups**

A broadcast group is the means by which a server broadcasts connectors over the network. A connector defines a way in which a client (or other server) can make connections to the server. For more information on what a connector is, please see Configuring the Transport.

The broadcast group takes a connector and broadcasts it on the network. Depending on which broadcasting technique you configure, it uses either UDP or JGroups to broadcast connector information.

Broadcast groups are defined in the server configuration file broker.xml. There can be many broadcast groups per Apache ActiveMQ Artemis server. All broadcast groups must be defined in a
Let's take a look at an example broadcast group from `broker.xml` that defines a UDP broadcast group:

```xml
<broadcast-groups>
  <broadcast-group name="my-broadcast-group">
    <local-bind-address>172.16.9.3</local-bind-address>
    <local-bind-port>5432</local-bind-port>
    <group-address>231.7.7.7</group-address>
    <group-port>9876</group-port>
    <broadcast-period>2000</broadcast-period>
    <connector-ref>netty-connector</connector-ref>
  </broadcast-group>
</broadcast-groups>
```

Some of the broadcast group parameters are optional and you'll normally use the defaults, but we specify them all in the above example for clarity. Let's discuss each one in turn:

**name**
- attribute. Each broadcast group in the server must have a unique name.

**local-bind-address**
- This is the local bind address that the datagram socket is bound to. If you have multiple network interfaces on your server, you would specify which one you wish to use for broadcasts by setting this property. If this property is not specified then the socket will be bound to the wildcard address, an IP address chosen by the kernel. This is a UDP specific attribute.

**local-bind-port**
- If you want to specify a local port to which the datagram socket is bound you can specify it here. Normally you would just use the default value of `-1` which signifies that an anonymous port should be used. This parameter is always specified in conjunction with `local-bind-address`. This is a UDP specific attribute.

**group-address**
- This is the multicast address to which the data will be broadcast. It is a class D IP address in the range `224.0.0.0` to `239.255.255.255`, inclusive. The address `224.0.0.0` is reserved and is not available for use. This parameter is mandatory. This is a UDP specific attribute.

**group-port**
- This is the UDP port number used for broadcasting. This parameter is mandatory. This is a UDP specific attribute.

**broadcast-period**
- This is the period in milliseconds between consecutive broadcasts. This parameter is optional, the default value is `2000` milliseconds.
This specifies the connector and optional backup connector that will be broadcasted (see Configuring the Transport for more information on connectors).

Here is another example broadcast group that defines a JGroups broadcast group:

```xml
<broadcast-groups>
  <broadcast-group name="my-broadcast-group">
    <broadcast-period>2000</broadcast-period>
    <jgroups-file>test-jgroups-file_ping.xml</jgroups-file>
    <jgroups-channel>activemq_broadcast_channel</jgroups-channel>
    <connector-ref>netty-connector</connector-ref>
  </broadcast-group>
</broadcast-groups>
```

To be able to use JGroups to broadcast, one must specify two attributes, i.e. `jgroups-file` and `jgroups-channel`, as discussed in details as following:

**jgroups-file**

attribute. This is the name of JGroups configuration file. It will be used to initialize JGroups channels. Make sure the file is in the java resource path so that Apache ActiveMQ Artemis can load it. The typical location for the file is the `etc` directory from the broker instance.

**jgroups-channel**

attribute. The name that JGroups channels connect to for broadcasting.

The JGroups attributes (`jgroups-file` and `jgroups-channel`) and UDP specific attributes described above are exclusive of each other. Only one set can be specified in a broadcast group configuration. Don't mix them!

The following is an example of a JGroups file

```xml
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="urn:org:jgroups"
  xsi:schemalocation="urn:org:jgroups
http://www.jgroups.org/schema/jgroups.xsd">
  <TCP bind_addr="$\{jgroups.bind_addr:site_local}\"
    bind_port="$\{jgroups.bind_port:7800}\"
    external_addr="$\{jgroups.external_addr}\"
    external_port="$\{jgroups.external_port}\"
    thread_pool.min_threads="0"
    thread_pool.max_threads="200"
    thread_pool.keep_alive_time="30000"/>
  <RED/>

  <!-- a location that can be found by both server's running -->
  <FILE_PING location="../file.ping.dir"/>
  <MERGE3 min_interval="10000"/>
</config>
```
As it shows, the file content defines a jgroups protocol stacks. If you want Apache ActiveMQ Artemis to use this stacks for channel creation, you have to make sure the value of `<jgroups-file>` in your broadcast-group/discovery-group configuration to be the name of this jgroups configuration file. For example if the above stacks configuration is stored in a file named "jgroups-stacks.xml" then your `<jgroups-file>` should be like

```
<jgroups-file>jgroups-stacks.xml</jgroups-file>
```

### Discovery Groups

While the broadcast group defines how connector information is broadcasted from a server, a discovery group defines how connector information is received from a broadcast endpoint (a UDP multicast address or JGroup channel).

A discovery group maintains a list of connector pairs - one for each broadcast by a different server. As it receives broadcasts on the broadcast endpoint from a particular server it updates its entry in the list for that server.

If it has not received a broadcast from a particular server for a length of time it will remove that server's entry from its list.

Discovery groups are used in two places in Apache ActiveMQ Artemis:

- By cluster connections so they know how to obtain an initial connection to download the topology
- By messaging clients so they know how to obtain an initial connection to download the topology

Although a discovery group will always accept broadcasts, its current list of available primary and backup servers is only ever used when an initial connection is made, from then server discovery is
done over the normal Apache ActiveMQ Artemis connections.

Each discovery group must be configured with broadcast endpoint (UDP or JGroups) that matches its broadcast group counterpart. For example, if broadcast is configured using UDP, the discovery group must also use UDP, and the same multicast address.

Defining Discovery Groups on the Server

For cluster connections, discovery groups are defined in the server side configuration file `broker.xml`. All discovery groups must be defined inside a `discovery-groups` element. There can be many discovery groups defined by Apache ActiveMQ Artemis server. Let’s look at an example:

```
<discovery-groups>
  <discovery-group name="my-discovery-group">
    <local-bind-address>172.16.9.7</local-bind-address>
    <group-address>231.7.7.7</group-address>
    <group-port>9876</group-port>
    <refresh-timeout>10000</refresh-timeout>
  </discovery-group>
</discovery-groups>
```

We’ll consider each parameter of the discovery group:

**name**

attribute. Each discovery group must have a unique name per server.

**local-bind-address**

If you are running with multiple network interfaces on the same machine, you may want to specify that the discovery group listens only a specific interface. To do this you can specify the interface address with this parameter. This parameter is optional. This is a UDP specific attribute.

**group-address**

This is the multicast IP address of the group to listen on. It should match the `group-address` in the broadcast group that you wish to listen from. This parameter is mandatory. This is a UDP specific attribute.

**group-port**

This is the UDP port of the multicast group. It should match the `group-port` in the broadcast group that you wish to listen from. This parameter is mandatory. This is a UDP specific attribute.

**refresh-timeout**

This is the period the discovery group waits after receiving the last broadcast from a particular server before removing that servers connector pair entry from its list. You would normally set this to a value significantly higher than the `broadcast-period` on the broadcast group otherwise servers might intermittently disappear from the list even though they are still broadcasting due
to slight differences in timing. This parameter is optional, the default value is 10000 milliseconds (10 seconds).

Here is another example that defines a JGroups discovery group:

```xml
<discovery-groups>
  <discovery-group name="my-broadcast-group">
    <jgroups-file>test-jgroups-file_ping.xml</jgroups-file>
    <jgroups-channel>activemq_broadcast_channel</jgroups-channel>
    <refresh-timeout>10000</refresh-timeout>
  </discovery-group>
</discovery-groups>
```

To receive broadcast from JGroups channels, one must specify two attributes, `jgroups-file` and `jgroups-channel`, as discussed in details as following:

**jgroups-file**
attribute. This is the name of JGroups configuration file. It will be used to initialize JGroups channels. Make sure the file is in the java resource path so that Apache ActiveMQ Artemis can load it.

**jgroups-channel**
attribute. The name that JGroups channels connect to for receiving broadcasts.

The JGroups attributes (`jgroups-file` and `jgroups-channel`) and UDP specific attributes described above are exclusive of each other. Only one set can be specified in a discovery group configuration. Don’t mix them!

**Discovery Groups on the Client Side**

Let’s discuss how to configure an Apache ActiveMQ Artemis client to use discovery to discover a list of servers to which it can connect. The way to do this differs depending on whether you’re using JMS or the core API.

**Configuring client discovery**

Use the `udp` URL scheme and a host:port combination matches the group-address and group-port from the corresponding `broadcast-group` on the server:

```
udp://231.7.7.7:9876
```

Connections created using this URI will be load-balanced across the list of servers that the discovery group maintains by listening on the multicast address specified in the discovery group configuration.

The aforementioned `refreshTimeout` parameter can be set directly in the URI.

There is also a URL parameter named `initialWaitTimeout`. If the corresponding JMS connection
factory or core session factory is used immediately after creation then it may not have had enough time to received broadcasts from all the nodes in the cluster. On first usage, the connection factory will make sure it waits this long since creation before creating the first connection. The default value for this parameter is 10000 milliseconds.

### 32.3.2. Discovery using static Connectors

Sometimes it may be impossible to use UDP on the network you are using. In this case it's possible to configure a connection with an initial list of possible servers. This could be just one server that you know will always be available or a list of servers where at least one will be available.

This doesn't mean that you have to know where all your servers are going to be hosted, you can configure these servers to use the reliable servers to connect to. Once they are connected their connection details will be propagated via the server it connects to

**Configuring a Cluster Connection**

For cluster connections there is no extra configuration needed, you just need to make sure that any connectors are defined in the usual manner, (see Configuring the Transport for more information on connectors). These are then referenced by the cluster connection configuration.

**Configuring a Client Connection**

A static list of possible servers can also be used by a normal client.

**Configuring client discovery**

A list of servers to be used for the initial connection attempt can be specified in the connection URI using a syntax with `()`, e.g.:

```
(tcp://myhost:61616,tcp://myhost2:61616)?reconnectAttempts=5
```

The brackets are expanded so the same query can be appended after the last bracket for ease.

### 32.4. Server-Side Message Load Balancing

If cluster connections are defined between nodes of a cluster, then Apache ActiveMQ Artemis will load balance messages arriving at a particular node from a client.

Let's take a simple example of a cluster of four nodes A, B, C, and D arranged in a symmetric cluster (described in Symmetrical Clusters section). We have a queue called `OrderQueue` deployed on each node of the cluster.

We have client Ca connected to node A, sending orders to the server. We have also have order processor clients Pa, Pb, Pc, and Pd connected to each of the nodes A, B, C, D. If no cluster connection was defined on node A, then as order messages arrive on node A they will all end up in the `OrderQueue` on node A, so will only get consumed by the order processor client attached to node A, Pa.
If we define a cluster connection on node A, then as ordered messages arrive on node A instead of all of them going into the local OrderQueue instance, they are distributed in a round-robin fashion between all the nodes of the cluster. The messages are forwarded from the receiving node to other nodes of the cluster. This is all done on the server side, the client maintains a single connection to node A.

For example, messages arriving on node A might be distributed in the following order between the nodes: B, D, C, A, B, D, C, A, B, D. The exact order depends on the order the nodes started up, but the algorithm used is round robin.

Apache ActiveMQ Artemis cluster connections can be configured to always blindly load balance messages in a round robin fashion irrespective of whether there are any matching consumers on other nodes, but they can be a bit cleverer than that and also be configured to only distribute to other nodes if they have matching consumers. We'll look at both these cases in turn with some examples, but first we'll discuss configuring cluster connections in general.

### 32.4.1. Configuring Cluster Connections

Cluster connections group servers into clusters so that messages can be load balanced between the nodes of the cluster. Let's take a look at a typical cluster connection. Cluster connections are always defined in broker.xml inside a cluster-connection element. There can be zero or more cluster connections defined per Apache ActiveMQ Artemis server.

```xml
<cluster-connections>
  <cluster-connection name="my-cluster">
    <address></address>
    <connector-ref>netty-connector</connector-ref>
    <check-period>1000</check-period>
    <connection-ttl>5000</connection-ttl>
    <min-large-message-size>5000</min-large-message-size>
    <call-timeout>5000</call-timeout>
    <retry-interval>500</retry-interval>
    <retry-interval-multiplier>1.0</retry-interval-multiplier>
    <max-retry-interval>5000</max-retry-interval>
    <initial-connect-attempts>-1</initial-connect-attempts>
    <reconnect-attempts>-1</reconnect-attempts>
    <use-duplicate-detection>true</use-duplicate-detection>
    <message-load-balancing>ON_DEMAND</message-load-balancing>
    <max-hops>1</max-hops>
    <confirmation-window-size>32000</confirmation-window-size>
    <call-failover-timeout>30000</call-failover-timeout>
    <notification-interval>1000</notification-interval>
    <notification-attempts>2</notification-attempts>
    <discovery-group-ref discovery-group-name="my-discovery-group"/>
  </cluster-connection>
</cluster-connections>
```

In the above cluster connection all parameters have been explicitly specified. The following shows all the available configuration options.
address

Each cluster connection only applies to addresses that match the specified address field. An address is matched on the cluster connection when it begins with the string specified in this field. The address field on a cluster connection also supports comma separated lists and an exclude syntax !. To prevent an address from being matched on this cluster connection, prepend a cluster connection address string with !.

In the case shown above the cluster connection will load balance messages sent to all addresses (since it’s empty).

The address can be any value and you can have many cluster connections with different values of address, simultaneously balancing messages for those addresses, potentially to different clusters of servers. By having multiple cluster connections on different addresses a single Apache ActiveMQ Artemis Server can effectively take part in multiple clusters simultaneously.

Be careful not to have multiple cluster connections with overlapping values of address, e.g. “europe” and “europe.news” since this could result in the same messages being distributed between more than one cluster connection, possibly resulting in duplicate deliveries.

Examples:

- ‘eu’ matches all addresses starting with ‘eu’
- ‘!eu’ matches all address except for those starting with ‘eu’
- ‘eu.uk,eu.de’ matches all addresses starting with either ‘eu.uk’ or ‘eu.de’
- ‘eu,!eu.uk’ matches all addresses starting with ‘eu’ but not those starting with ‘eu.uk’

- Address exclusion will always takes precedence over address inclusion.
- Address matching on cluster connections does not support wild-card matching.

connector-ref

This is the connector which will be sent to other nodes in the cluster so they have the correct cluster topology.

This parameter is mandatory.

check-period

The period (in milliseconds) used to check if the cluster connection has failed to receive pings from another server. Default is 30000.

connection-ttl

This is how long a cluster connection should stay alive if it stops receiving messages from a specific node in the cluster. Default is 60000.

min-large-message-size

If the message size (in bytes) is larger than this value then it will be split into multiple segments when sent over the network to other cluster members. Default is 102400.
**call-timeout**

When a packet is sent via a cluster connection and is a blocking call, i.e. for acknowledgements, this is how long it will wait (in milliseconds) for the reply before throwing an exception. Default is 30000.

**retry-interval**

We mentioned before that, internally, cluster connections cause bridges to be created between the nodes of the cluster. If the cluster connection is created and the target node has not been started, or say, is being rebooted, then the cluster connections from other nodes will retry connecting to the target until it comes back up, in the same way as a bridge does.

This parameter determines the interval in milliseconds between retry attempts. It has the same meaning as the `retry-interval` on a bridge (as described in Core Bridges).

This parameter is optional and its default value is 500 milliseconds.

**retry-interval-multiplier**

This is a multiplier used to increase the `retry-interval` after each reconnect attempt, default is 1.

**max-retry-interval**

The maximum delay (in milliseconds) for retries. Default is 2000.

**initial-connect-attempts**

The number of times the system will try to connect a node in the cluster initially. If the max-retry is achieved this node will be considered permanently down and the system will not route messages to this node. Default is -1 (infinite retries).

**reconnect-attempts**

The number of times the system will try to reconnect to a node in the cluster. If the max-retry is achieved this node will be considered permanently down and the system will stop routing messages to this node. Default is -1 (infinite retries).

**use-duplicate-detection**

Internally cluster connections use bridges to link the nodes, and bridges can be configured to add a duplicate id property in each message that is forwarded. If the target node of the bridge crashes and then recovers, messages might be resent from the source node. By enabling duplicate detection any duplicate messages will be filtered out and ignored on receipt at the target node.

This parameter has the same meaning as `use-duplicate-detection` on a bridge. For more information on duplicate detection, please see Duplicate Detection. Default is true.

**message-load-balancing**

This parameter determines if/how messages will be distributed between other nodes of the cluster. It can be one of four values - OFF, STRICT, OFF_WITH_REdistribution or ON_DEMAND (default). This parameter replaces the deprecated forward-when-no-consumers parameter.

If this is set to OFF then messages will never be forwarded to another node in the cluster
If this is set to **STRICT** then each incoming message will be round robin'd even though the same queues on the other nodes of the cluster may have no consumers at all, or they may have consumers that have non matching message filters (selectors). Note that Apache ActiveMQ Artemis will *not* forward messages to other nodes if there are no queues of the same name on the other nodes, even if this parameter is set to **STRICT**. Using **STRICT** is like setting the legacy `forward-when-no-consumers` parameter to **true**.

If this is set to **ON_DEMAND** then Apache ActiveMQ Artemis will only forward messages to other nodes of the cluster if the address to which they are being forwarded has queues which have consumers, and if those consumers have message filters (selectors) at least one of those selectors must match the message. Using **ON_DEMAND** is like setting the legacy `forward-when-no-consumers` parameter to **false**.

If this is set to **OFF_WITH_REDUCTION** then like with 'OFF' messages won’t be initially routed to other nodes in the cluster. However, if redistribution is configured, it can forward messages in the normal way. In this way local consumers will always have priority.

Keep in mind that this message forwarding/balancing is what we call "initial distribution." It is different than redistribution which is discussed below.

Default is **ON_DEMAND**.

**max-hops**

When a cluster connection decides the set of nodes to which it might load balance a message, those nodes do not have to be directly connected to it via a cluster connection. Apache ActiveMQ Artemis can be configured to also load balance messages to nodes which might be connected to it only indirectly with other Apache ActiveMQ Artemis servers as intermediates in a chain.

This allows Apache ActiveMQ Artemis to be configured in more complex topologies and still provide message load balancing. We’ll discuss this more later in this chapter.

The default value for this parameter is **1**, which means messages are only load balanced to other Apache ActiveMQ Artemis serves which are directly connected to this server. This parameter is optional.

**confirmation-window-size**

The size (in bytes) of the window used for sending confirmations from the server connected to. So once the server has received `confirmation-window-size` bytes it notifies its client, default is 1048576. A value of -1 means no window.

**producer-window-size**

The size for producer flow control over cluster connection. It's by default is 1MB.

**call-failover-timeout**

Similar to `call-timeout` but used when a call is made during a failover attempt. Default is -1 (no timeout).

**notification-interval**

How often (in milliseconds) the cluster connection should broadcast itself when attaching to the
cluster. Default is 1000.

**notification-attempts**
How many times the cluster connection should broadcast itself when connecting to the cluster. Default is 2.

**discovery-group-ref**
This parameter determines which discovery group is used to obtain the list of other servers in the cluster that this cluster connection will make connections to.

Alternatively if you would like your cluster connections to use a static list of servers for discovery then you can do it like this.

```xml
<cluster-connection name="my-cluster">
  ...
  <static-connectors>
    <connector-ref>server0-connector</connector-ref>
    <connector-ref>server1-connector</connector-ref>
  </static-connectors>
</cluster-connection>
```

Here we have defined 2 servers that we know for sure will that at least one will be available. There may be many more servers in the cluster but these will; be discovered via one of these connectors once an initial connection has been made.

### 32.4.2. Cluster User Credentials

When creating connections between nodes of a cluster to form a cluster connection, Apache ActiveMQ Artemis uses a cluster user and cluster password which is defined in `broker.xml`:

```xml
<cluster-user>ACTIVEMQ.CLUSTER.ADMIN.USER</cluster-user>
<cluster-password>CHANGE ME!!</cluster-password>
```

It is imperative that these values are changed from their default, or remote clients will be able to make connections to the server using the default values. If they are not changed from the default, Apache ActiveMQ Artemis will detect this and pester you with a warning on every start-up.

### 32.5. Client-Side Load balancing

With Apache ActiveMQ Artemis client-side load balancing, subsequent sessions created using a single session factory can be connected to different nodes of the cluster. This allows sessions to spread smoothly across the nodes of a cluster and not be "clumped" on any particular node.

The load balancing policy to be used by the client factory is configurable. Apache ActiveMQ Artemis provides four out-of-the-box load balancing policies, and you can also implement your own and use
The out-of-the-box policies are

- **Round Robin.** With this policy the first node is chosen randomly then each subsequent node is chosen sequentially in the same order.

  For example nodes might be chosen in the order B, C, D, A, B, C, D, A, B, C, D or C, D, A, B, C, D, A, B, C.

  Use

  ```java
  org.apache.activemq.artemis.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy
  ```

  as the `<connection-load-balancing-policy-class-name>`.

- **Random.** With this policy each node is chosen randomly.

  Use

  ```java
  org.apache.activemq.artemis.api.core.client.loadbalance.RandomConnectionLoadBalancingPolicy
  ```

  as the `<connection-load-balancing-policy-class-name>`.

- **Random Sticky.** With this policy the first node is chosen randomly and then re-used for subsequent connections.

  Use

  ```java
  org.apache.activemq.artemis.api.core.client.loadbalance.RandomStickyConnectionLoadBalancingPolicy
  ```

  as the `<connection-load-balancing-policy-class-name>`.

- **First Element.** With this policy the “first” (i.e. 0th) node is always returned.

  Use

  ```java
  org.apache.activemq.artemis.api.core.client.loadbalance.FirstElementConnectionLoadBalancingPolicy
  ```

  as the `<connection-load-balancing-policy-class-name>`.

You can also implement your own policy by implementing the interface

```java
org.apache.activemq.artemis.api.core.client.loadbalance.ConnectionLoadBalancingPolicy
```

Specifying which load balancing policy to use differs whether you are using JMS or the core API. If you don’t specify a policy then the default will be used which is

```java
org.apache.activemq.artemis.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy
```

The parameter `connectionLoadBalancingPolicyClassName` can be set on the URI to configure what load balancing policy to use:

```text
```

The set of servers over which the factory load balances can be determined in one of two ways:

- **Specifying servers explicitly in the URL.** This also requires setting the `useTopologyForLoadBalancing` parameter to `false` on the URL.
32.6. Specifying Members of a Cluster Explicitly

Sometimes you want to explicitly define a cluster more explicitly, that is control which server connect to each other in the cluster. This is typically used to form non symmetrical clusters such as chain cluster or ring clusters. This can only be done using a static list of connectors and is configured as follows:

```xml
<cluster-connection name="my-cluster">
    <address/>
    <connector-ref>netty-connector</connector-ref>
    <retry-interval>500</retry-interval>
    <use-duplicate-detection>true</use-duplicate-detection>
    <message-load-balancing>STRICT</message-load-balancing>
    <max-hops>1</max-hops>
    <static-connectors allow-direct-connections-only="true">
        <connector-ref>server1-connector</connector-ref>
    </static-connectors>
</cluster-connection>
```

In this example we have set the attribute `allow-direct-connections-only` which means that the only server that this server can create a cluster connection to is server1-connector. This means you can explicitly create any cluster topology you want.

32.7. Message Redistribution

Another important part of clustering is message redistribution. Earlier we learned how server side message load balancing round robins messages across the cluster. If `message-load-balancing` is OFF or ON_DEMAND then messages won’t be forwarded to nodes which don’t have matching consumers. This is great and ensures that messages aren’t moved to a queue which has no consumers to consume them. However, there is a situation it doesn’t solve: What happens if the consumers on a queue close after the messages have been sent to the node? If there are no consumers on the queue the message won’t get consumed and we have a starvation situation.

This is where message redistribution comes in. With message redistribution Apache ActiveMQ Artemis can be configured to automatically redistribute messages from queues which have no consumers or consumers with filters that don’t match messages. The messages are re-routed to other nodes in the cluster which do have matching consumers. To enable this functionality `message-load-balancing` must be ON_DEMAND or OFF_WITH_REDISTRIBUTION

Message redistribution can be configured to kick in immediately after the need to redistribute is detected, or to wait a configurable delay before redistributing. By default, message redistribution is disabled.

Message redistribution can be configured on a per address basis, by specifying the redistribution delay in the address settings. For more information on configuring address settings, please see
Configuring Addresses and Queues via Address Settings.

Here's an address settings snippet from broker.xml showing how message redistribution is enabled for a set of queues:

```
<address-settings>
    <address-setting match="#">
        <redistribution-delay>0</redistribution-delay>
    </address-setting>
</address-settings>
```

The above address-settings block would set a redistribution-delay of 0 for any queue which is bound to any address. So the above would enable instant (no delay) redistribution for all addresses.

The attribute match can be an exact match or it can be a string that conforms to the Apache ActiveMQ Artemis wildcard syntax (described in Wildcard Syntax).

The element redistribution-delay defines the delay in milliseconds between detecting the need for redistribution and actually attempting redistribution. A delay of zero means the messages will be immediately redistributed. A value of -1 signifies that messages will never be redistributed. The default value is -1.

It often makes sense to introduce a delay before redistributing as it's a common case that a consumer closes but another one quickly is created on the same queue, in such a case you probably don't want to redistribute immediately since the new consumer will arrive shortly.

### 32.8. Cluster topologies

Apache ActiveMQ Artemis clusters can be connected together in many different topologies, let's consider the two most common ones here

#### 32.8.1. Symmetric cluster

A symmetric cluster is probably the most common cluster topology.

With a symmetric cluster every node in the cluster is connected to every other node in the cluster. In other words every node in the cluster is no more than one hop away from every other node.

To form a symmetric cluster every node in the cluster defines a cluster connection with the attribute max-hops set to 1. Typically the cluster connection will use server discovery in order to know what other servers in the cluster it should connect to, although it is possible to explicitly define each target server too in the cluster connection if, for example, UDP is not available on your network.

With a symmetric cluster each node knows about all the queues that exist on all the other nodes and what consumers they have. With this knowledge it can determine how to load balance and redistribute messages around the nodes.

Don't forget this warning when creating a symmetric cluster.
32.8.2. Chain cluster

With a chain cluster, each node in the cluster is not connected to every node in the cluster directly, instead the nodes form a chain with a node on each end of the chain and all other nodes just connecting to the previous and next nodes in the chain.

An example of this would be a three node chain consisting of nodes A, B and C. Node A is hosted in one network and has many producer clients connected to it sending order messages. Due to corporate policy, the order consumer clients need to be hosted in a different network, and that network is only accessible via a third network. In this setup node B acts as a mediator with no producers or consumers on it. Any messages arriving on node A will be forwarded to node B, which will in turn forward them to node C where they can get consumed. Node A does not need to directly connect to C, but all the nodes can still act as a part of the cluster.

To set up a cluster in this way, node A would define a cluster connection that connects to node B, and node B would define a cluster connection that connects to node C. In this case we only want cluster connections in one direction since we're only moving messages from node A->B->C and never from C->B->A.

For this topology we would set max-hops to 2. With a value of 2 the knowledge of what queues and consumers that exist on node C would be propagated from node C to node B to node A. Node A would then know to distribute messages to node B when they arrive, even though node B has no consumers itself, it would know that a further hop away is node C which does have consumers.

32.8.3. Scaling Down

Apache ActiveMQ Artemis supports scaling down a cluster with no message loss (even for non-durable messages). This is especially useful in certain environments (e.g. the cloud) where the size of a cluster may change relatively frequently. When scaling up a cluster (i.e. adding nodes) there is no risk of message loss, but when scaling down a cluster (i.e. removing nodes) the messages on those nodes would be lost unless the broker sent them to another node in the cluster. Apache ActiveMQ Artemis can be configured to do just that.

To enable this behavior configure scale-down in the live-only ha-policy, e.g.:

```xml
<ha-policy>
  <live-only>
    <scale-down>
      <enabled>true</enabled>
      <discovery-group-ref discovery-group-name="my-discovery-group"/>
    </scale-down>
  </live-only>
</ha-policy>
```

If scale-down/enabled is true then when the server is shutdown gracefully (i.e. stopped without crashing) it will find another node in the cluster and send all of its messages (both durable and non-durable) to that node. The messages are processed in order and go to the back of the respective queues on the other node (just as if the messages were sent from an external client for the first
The target of the scale down operation can be configured a few different ways. The above example uses discovery-group-ref to reference a discovery-group which will be used to find the target broker. This should be the same discovery-group referenced by your cluster-connection. You can also specify a static list of connector elements, e.g.:

```
<connectors>
  ...
  <connector name="server0-connector">tcp://server0:61616</connector>
</connectors>
...

<ha-policy>
  <live-only>
    <scale-down>
      <enabled>true</enabled>
      <connectors>
        <connector-ref>server0-connector</connector-ref>
      </connectors>
    </scale-down>
  </live-only>
</ha-policy>
```

It's also possible to specify group-name. If this is specified then messages will only be sent to another node in the cluster that uses the same group-name as the server being shutdown, e.g.:

```
<ha-policy>
  <live-only>
    <scale-down>
      <enabled>true</enabled>
      <group-name>my-group</group-name>
      <discovery-group-ref discovery-group-name="my-discovery-group"/>
    </scale-down>
  </live-only>
</ha-policy>
```

If cluster nodes are grouped together with different group-name values beware. If all the nodes in a single group are shut down then the messages from that node/group will be lost.
Chapter 33. Federation

Federation allows transmission of messages between brokers without requiring clustering.

A federated address can replicate messages published from an upstream address to a local address. n.b. This is only supported with multicast addresses.

A federated queue lets a local consumer receive messages from an upstream queue.

A broker can contain federated and local-only components - you don’t need to federate everything if you don’t want to.

33.1. Benefits

33.1.1. WAN

The source and target servers do not have to be in the same cluster which makes federation suitable for reliably sending messages from one cluster to another, for instance across a WAN, between cloud regions or where connection may be unreliable.

Federation has built-in resilience to failure so if the target server connection is lost, e.g. due to network failure, federation will retry connecting to the target until it comes back online. When it comes back online it will resume operation as normal.

33.1.2. Loose Coupling of Brokers

Federation can transmit messages between brokers (or clusters) in different administrative domains:

• they may have different configuration, users and setup;
• they may run on different versions of ActiveMQ Artemis

33.1.3. Dynamic and Selective

Federation is applied by policies, that match address and queue names, and then apply.

This means that federation can dynamically be applied as queues or addresses are added and removed, without need to individually configure them.

Likewise, policies are selective. They apply with multiple include and exclude matches.

Multiple policies can be applied directly to multiple upstreams. Policies can be grouped into policy sets and then applied to upstreams to make managing easier.

33.2. Address Federation

Address federation is similar to full multicast over the connected brokers. Every message sent to address on Broker-A will be delivered to every queue on that broker, but also will be delivered to
Broker-B and all their attached queues.

Address Federation

For further details please see Address Federation.

33.3. Queue Federation

Federated queues act as a single logical queue with multiple receivers on multiple machines. They can be used for load balancing. If brokers are in the same Availability Zone you would look to cluster them. The advantage of queue federation is that it does not require clustering, so it is suitable for over WAN, cross-region or on/off-premises usage.

Queue Federation
For further details please see Queue Federation.

33.4. WAN Full Mesh

It is also possible to provide a WAN mesh of brokers, which can

- replicate by Address Federation
- route and load balance using Queue Federation
- link distant producers and consumers

*Example of possible full federation mesh*
33.5. Configuring Federation

Example configuration in `broker.xml`:

```xml
<federations>
  <federation name="eu-north-1-federation">
    <upstream name="eu-west-1" user="westuser" password="32a10275cf4ab4e9">
      <static-connectors>
        <connector-ref>connector1</connector-ref>
      </static-connectors>
      <policy ref="policySetA"/>
    </upstream>
    <upstream name="eu-east-1" user="eastuser" password="32a10275cf4ab4e9">
      <discovery-group-ref discovery-group-name="ue-west-dg"/>
      <policy ref="policySetA"/>
    </upstream>
  </federation>
  <policy-set name="policySetA">
    <policy ref="address-federation" />  
    <policy ref="queue-federation" />  
  </policy-set>
  <queue-policy name="queue-federation">
    <exclude queue-match="federated_queue" address-match="#"/>
  </queue-policy>
  <address-policy name="address-federation">
    <include address-match="federated_address"/>
  </address-policy>
</federations>
```
In the above example we have shown the basic key parameters needed to configure federation for a queue and address to multiple upstream.

The example shows a broker eu-north-1 connecting to two upstream brokers eu-east-1 and eu-west-1. Applying queue federation to queue federated_queue and address federation to address federated_address.

**It is important that federation name is globally unique.**

There are many configuration options that you can apply. These are detailed in the individual docs:

- Address Federation
- Queue Federation

Extra parameters from the URI of a connector-ref can be used to override or provide additional configuration to the ServiceLocator.

### 33.5.1. Large Messages

If federation has to process large messages, the default `ackBatchSize` and `consumerWindowSize` for the consumer will need to be changed to limit the number of in-flight messages and to enable large message flow. These options can be supplied as parameters on the referenced connector URI, for example:

```text
tcp://<host>:<port>?ackBatchSize=100&consumerWindowSize=-1
```
Chapter 34. Address Federation

Address federation is similar to full multicast over the connected brokers. Every message sent to address on Broker-A will be delivered to every queue on that broker, but also will be delivered to Broker-B and all their attached queues.

Address federation dynamically links to other addresses in upstream or downstream brokers. It automatically creates a queue on the remote address for itself, which it then consumes and copies to the local address, as if they were published directly to it.

Upstream brokers or addresses do not need to be reconfigured. Just add the needed permissions to the address for the downstream broker. The same applies for downstream configurations.

34.1. Topology Patterns

34.1.1. Symmetric
As seen above, a publisher and consumer are connected to each broker. Queues (consumers of those queues) can receive messages published by either publisher.

In this setup, it is important to set `max-hops=1`, so messages are copied only once and cyclic replication is avoided. If `max-hops` is not configured correctly, consumers will get multiple copies of the same message.

### 34.1.2. Full Mesh
Figure 3. Address Federation - Full Mesh

This setup is identical to symmetric above. All brokers are symmetrically federating each other, creating a full mesh.

As illustrated, a publisher and consumer are connected to each broker. Queues and thus consumers on those queues, can receive messages published by either publisher.

Again, it is important to set \texttt{max-hops=1}, so messages are copied only once and cyclic replication is avoided. If \texttt{max-hops} is not configured correctly, consumers will get multiple copies of the same message.

34.1.3. Ring
In a ring of brokers each federated address is upstream to just one other in the ring. To avoid the cyclic issue, it is important to set $\text{max-hops}$ to $n - 1$ where $n$ is the number of nodes in the ring. In the example above, property is set to 5, so that every address in the ring receives the message exactly once.

Whilst this setup is cheap in regard to connections it is brittle. If a single broker fails then the ring fails.

34.1.4. Fan out
Figure 5. Address Federation - Fan Out

One main address (requires no configuration) is linked to a tree of downstream federated addresses. The tree can extend to any depth and can be extended further without needing to reconfigure existing brokers.

In this case, messages published to the main address can be received by any consumer connected to any broker in the tree.

34.1.5. Divert Binding Support

Divert binding support can be added as part of the address policy configuration. This will allow the federation to respond to divert bindings to create demand. For example, let's say there is one address called `test.federation.source` that is included as a match for the federated address and another address called `test.federation.target` that is not included. Normally, when a queue is created on `test.federation.target`, this would not cause a federated consumer to be created because the address is not part of the included matches. However, if we create a divert binding such that `test.federation.source` is the source address and `test.federation.target` is the forwarded address then demand will now be created. The source address still must be `multicast` but the target address can be `multicast` or `anycast`.
An example use case for this might be a divert, that redirects JMS topics (multicast addresses) to a JMS queue (anycast addresses). This allows message load balancing on a topic for legacy consumers not supporting JMS 2.0 and shared subscriptions.

### 34.2. Configuring Address Federation

Federation is configured in `broker.xml` file.

Sample Address Federation setup:

```xml
<federations>
  <federation name="eu-north-1" user="federation_username"
              password="32a10275cf4ab4e9">
    <upstream name="eu-east-1">
      <static-connectors>
        <connector-ref>eu-east-connector1</connector-ref>
        <connector-ref>eu-east-connector2</connector-ref>
      </static-connectors>
      <policy ref="news-address-federation"/>
    </upstream>
    <upstream name="eu-west-1">
      <static-connectors>
        <connector-ref>eu-west-connector1</connector-ref>
        <connector-ref>eu-west-connector2</connector-ref>
      </static-connectors>
      <policy ref="news-address-federation"/>
    </upstream>
  </federation>
  <address-policy name="news-address-federation" max-hops="1"
                  auto-delete="true"
                  auto-delete-delay="300000"
                  auto-delete-message-count="-1"
                  transformer-ref="news-transformer">
    <include address-match="queue.bbc.new"/>
    <include address-match="queue.usatoday"/>
    <include address-match="queue.news.#"/>
    <exclude address-match="queue.news.sport.#"/>
  </address-policy>

  <transformer name="news-transformer">
    <class-name>org.foo.NewsTransformer</class-name>
    <property key="key1" value="value1"/>
    <property key="key2" value="value2"/>
  </transformer>
</federations>
```

In the above setup, downstream broker `eu-north-1` is configured to connect to two upstream brokers `eu-east-1` and `eu-west-1`. Credentials used for both connections to brokers in this sample are shared. Should they be different per upstream, you can alter credentials at the upstream level.
Both upstreams are configured with the same address-policy `news-address-federation`, that is selecting addresses which match any of the include criteria and exclude anything that starts `queue.news.sport`.

**It is important that federation name is globally unique.**

**address-policy parameters**

**name**

All address-policies must have a unique name in the server.

**include**

The address-match pattern to include addresses. Multiple of these can be set. If none are set all addresses are matched.

**exclude**

The address-match pattern to exclude addresses. Multiple of these can be set.

**max-hops**

The maximum number of hops that message can perform across federated addresses. See Topology Patterns above for details.

**auto-delete**

For address federation, the downstream dynamically creates a durable queue on the upstream address. This is used to mark if the upstream queue should be deleted once downstream disconnects and the delay and message count parameters have been met. It is useful in automating cleanup. You may wish to disable this, if you want messages to be queued for the downstream when disconnect no matter what.

**auto-delete-delay**

The amount of time in milliseconds after the downstream broker has disconnected, before the upstream queue can be eligible for `auto-delete`.

**auto-delete-message-count**

Maximum number of messages allowed in the upstream queue to be eligible for `auto-delete`, after the downstream broker has disconnected.

**transformer-ref**

Reference name of a transformer (see transformer config) that you may wish to configure to transform the message on federation transfer.

**enable-divert-bindings**

Setting to `true` will enable divert bindings to be listened for demand. If there is a divert binding with an address that matches the included addresses for the stream, any queue bindings that match the forward address of the divert will create demand. Default is `false`.

---

**address-policy** and **queue-policy** elements can be defined in the same federation and be linked to the same upstream.
**transformer parameters**

**name**
Unique name in the server used to reference the transformer in `address-policy` and `queue-policy`.

**transformer-class-name**
This is the name of a user-defined class which implements the `org.apache.activemq.artemis.core.server.transformer.Transformer` interface.

If specified, then the transformer's `transform()` method will be invoked with the message before it is transferred. This gives you the opportunity to transform the message's header or body before it is federated.

**property**
holds key-value pairs that can be used to configure the transformer.

---

**Upstream parameters**

Tag `upstream` defines the upstream broker connection and the policies to use.

**name**
Unique name for upstream federated server.

**user**
This optional attribute determines the username to use when creating the upstream connection to the remote server. If not specified, the shared federation user and password will be used if they were set.

**password**
This optional attribute determines the password to use when creating the upstream connection to the remote server. If not specified, the shared federation user and password will be used if they were set.

**static-connectors**
Either this or `discovery-group-ref` is used to connect the bridge to the target server.

The `static-connectors` is a list of `connector-ref` elements pointing to `connector` elements defined elsewhere. A `connector` encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc.) as well as the server connection parameters (host, port etc).

For more information about what connectors are and how to configure them, see Configuring the Transport.

**discovery-group-ref**
Either this or `static-connectors` is used to connect the bridge to the target server.

The `discovery-group-ref` element has one attribute - `discovery-group-name`. This attribute points to a `discovery-group` defined elsewhere. For more information about what `discovery-groups` are
and how to configure them, see *Discovery Groups*.

**ha**
Optional parameter determines whether this bridge should support high availability. Using `true` will connect to any available server in a cluster and support failover. The default value is `false`.

**circuit-breaker-timeout**
When a connection issue occurs, as the single connection is shared by many federated queue and address consumers, to avoid each one trying to reconnect and possibly causing a thundering herd issue, the first one will be tried. If unsuccessful the circuit breaker will open, returning the same exception to all connections. This is the timeout until the circuit can be closed and connection retried. Measured in milliseconds.

**share-connection**
If there is a downstream and upstream connection configured for the same broker then the same connection will be shared as long as both stream configs set this flag to true. Default is `false`.

**check-period**
The period (in milliseconds) used to check if the federation connection has failed to receive pings from another server. Default is `30000`.

**connection-ttl**
This is how long a federation connection should stay alive if it stops receiving messages from the remote broker. Default is `60000`.

**call-timeout**
When a packet is sent via a federation connection and is a blocking call, i.e. for acknowledgements, this is how long it will wait (in milliseconds) for the reply before throwing an exception. Default is `30000`.

**call-failover-timeout**
Similar to `call-timeout` but used when a call is made during a failover attempt. Default is `-1` (no timeout).

**retry-interval**
This optional parameter determines the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is `500` milliseconds.

**retry-interval-multiplier**
Used to increase the `retry-interval` after each reconnect attempt, default is `1`.

**max-retry-interval**
The maximum delay (in milliseconds) for retries. Default is `2000`.

**initial-connect-attempts**
The number of times the system will try to connect to the remote broker in the federation. If the
max is reached this broker will be considered permanently down and the system will not route messages to this broker. Default is -1 (infinite retries).

reconnect-attempts
The number of times the system will try to reconnect to the remote broker in the federation. If the max is reached this broker will be considered permanently down and the system will stop routing messages to this broker. Default is -1 (infinite retries).

34.3. Configuring Downstream Federation

Similarly to upstream configuration, a downstream configuration can be configured. This works by sending a command to the downstream broker to have it create an upstream connection back to the downstream broker. The benefit of this is being able to configure everything for federation on one broker in some cases to make it easier, such as a hub and spoke topology.

All the same configuration options apply to downstream as does upstream with the exception of one extra configuration flag that needs to be set:

upstream-connector-ref
Is an element pointing to a connector elements defined elsewhere. This reference is used to tell the downstream broker what connector to use to create a new upstream connection back to the downstream broker.

A connector encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc.) as well as the server connection parameters (host, port etc). For more information about what connectors are and how to configure them, see Configuring the Transport.

Sample Downstream Address Federation setup:

```xml
<!--Other config Here -->

<connectors>
  <connector name="netty-connector">tcp://localhost:61616</connector>
  <connector name="eu-west-1-connector">tcp://localhost:61616</connector>
  <connector name="eu-east-1-connector">tcp://localhost:61617</connector>
</connectors>

<acceptors>
  <acceptor name="netty-acceptor">tcp://localhost:61616</acceptor>
</acceptors>

<!--Other config Here -->

<federations>
  <federation name="eu-north-1" user="federation_username" password="32a10275cf4ab4e9">
    <downstream name="eu-east-1">
      <static-connectors>
        <connector-ref>eu-east-1-connector</connector-ref>
      </static-connectors>
    </downstream>
  </federation>
</federations>
```
<static-connectors>
  <upstream-connector-ref>netty-connector</upstream-connector-ref>
  <policy ref="news-address-federation"/>
</downstream>

<downstream name="eu-west-1">
  <static-connectors>
    <connector-ref>eu-west-1-connector</connector-ref>
  </static-connectors>
  <upstream-connector-ref>netty-connector</upstream-connector-ref>
  <policy ref="news-address-federation"/>
</downstream>

<address-policy name="news-address-federation" max-hops="1" auto-delete="true" auto-delete-delay="300000" auto-delete-message-count="-1" transformer-ref="news-transformer">
  <include address-match="queue.bbc.new"/>
  <include address-match="queue.usatoday"/>
  <include address-match="queue.news.#"/>
  <exclude address-match="queue.news.sport.#"/>
</address-policy>

<transformer name="news-transformer">
  <class-name>org.foo.NewsTransformer</class-name>
  <property key="key1" value="value1"/>
  <property key="key2" value="value2"/>
</transformer>
</federation>
Chapter 35. Queue Federation

This feature provides a way of balancing the load of a single queue across remote brokers.

A federated queue links to other queues (called upstream queues). It will retrieve messages from upstream queues in order to satisfy demand for messages from local consumers. The upstream queues do not need to be reconfigured and they do not have to be on the same broker or in the same cluster.

All of the configuration needed to establish the upstream links and the federated queue is in the downstream broker.

35.1. Use Cases

This is not an exhaustive list of what you can do with and the benefits of federated queues, but simply some ideas.

• Higher capacity

   By having a "logical" queue distributed over many brokers. Each broker would declare a federated queue with all the other federated queues upstream. (The links would form a complete bi-directional graph on n queues.)

By having this a logical distributed queue is capable of having a much higher capacity than a single queue on a single broker. When will perform best when there is some degree of locality.

   e.g. as many messages as possible are consumed from the same broker as they were published to, where federation only needs to move messages around in order to perform load balancing.

[federated queue symmetric] | images/federated-queue-symmetric.gif

Figure 6. Federated Queue Symmetric

• Supporting multi region or venue

   In a multi region setup you may have producers in one region or venue and the consumer in another. Typically you want producers and consumer to keep their connections local to the region, in such as case you can deploy brokers in each region where producers and consumer are, and use federation to move messages over the WAN between regions.

[federated queue] | images/federated-queue.gif

Figure 7. Federated Queue

• Communication between the secure enterprise lan and the DMZ.

   Where a number of producer apps maybe in the DMZ and a number of consumer apps in the secure enterprise lan, it may not suitable to allow the producers to connect through to the broker in the secure enterprise lan.

   In this scenario you could deploy a broker in the DMZ where the producers publish to, and then
have the broker in the enterprise lan connect out to the DMZ broker and federate the queues so that messages can traverse.

This is similar to supporting multi region or venue.

- Migrating between two clusters. Consumers and publishers can be moved in any order and the messages won’t be duplicated (which is the case if you do exchange federation). Instead, messages are transferred to the new cluster when your consumers are there. Here for such a migration with blue/green or canary moving a number of consumers on the same queue, you may want to set the priority-adjustment to 0, or even a positive value, so message would actively flow to the federated queue.

- Dual Federation - potential for messages to flip-flop between clusters. If the backlog on your queues exceeds the available local credit across consumers, any lower priority federation consumer becomes a candidate for dispatch and messages will be federated. Eventually all messages may migrate and the scenario can repeat on the other cluster. Applying a rate limit to the connector url can help mitigate but this could have an adverse effect on migration when there are no local consumers. To better support this use case, it is possible to configure the consumerWindowSize to zero on the referenced connector URL: tcp://<host>:<port>?consumerWindowSize=0. This will cause the federation consumer to pull messages in batches only when the local queue has excess capacity. This means that federation won’t ever drain more messages than it can handle, such that messages would flip-flop. The batch size is derived from the relevant address settings defaultConsumerWindowSize.

### 35.2. Configuring Queue Federation

Federation is configured in `broker.xml` file.

Sample Queue Federation setup:

```xml
<federations>
  <federation name="eu-north-1" user="federation_username" password="32a10275cf4ab4e9">
    <upstream name="eu-east-1">
      <static-connectors>
        <connector-ref>eu-east-connector1</connector-ref>
        <connector-ref>eu-east-connector2</connector-ref>
      </static-connectors>
      <policy ref="news-queue-federation"/>
    </upstream>
  </federation>
  <federation name="eu-west-1" user="federation_username" password="32a10275cf4ab4e9">
    <upstream name="eu-west-1">
      <static-connectors>
        <connector-ref>eu-west-connector1</connector-ref>
        <connector-ref>eu-west-connector2</connector-ref>
      </static-connectors>
      <policy ref="news-queue-federation"/>
    </upstream>
  </federation>
  <queue-policy name="news-queue-federation" priority-adjustment="-5" include-
In the above setup downstream broker eu-north-1 is configured to connect to two upstream brokers eu-east-1 and eu-west-1, the credentials used for both connections to both brokers in this sample are shared, you can set user and password at the upstream level should they be different per upstream.

Both upstreams are configured with the same queue-policy news-queue-federation, that is selecting addresses which match any of the include criteria, but will exclude any queues that end with .local, keeping these as local queues only.

It is important that federation name is globally unique.

Priority ordered queue-policy parameters

name
All address-policies must have a unique name in the server.

include
The address-match pattern to include addresses. Multiple of these can be set. If none are set all addresses are matched.

exclude
The address-match pattern to exclude addresses. Multiple of these can be set.

priority-adjustment
when a consumer attaches its priority is used to make the upstream consumer, but with an adjustment by default -1, so that local consumers get load balanced first over remote, this enables this to be configurable should it be wanted/needed.

include-federated
by default this is false, we do not federate a federated consumer, this is to avoid issue, where in symmetric or any closed loop setup you could end up when no "real" consumers attached with messages flowing round and round endlessly.
There is though a valid case that if you do not have a close loop setup e.g. three brokers in a
chain (A->B->C) with producer at broker A and consumer at C, you would want broker B to re-
federate the consumer onto A.

**transformer-ref**

The ref name for a transformer (see transformer config) that you may wish to configure to
transform the message on federation transfer.

---

*address-policy* and *queue-policy* elements are able to be defined in the same
federation, and they can be linked to the same upstream.

---

**Priority ordered transformer parameters**

- **name**
  
  Unique name in the server used to reference the transformer in *address-policy* and *queue-policy*

- **transformer-class-name**
  
  An optional *transformer-class-name* can be specified. This is the name of a user-defined class
  which implements the `org.apache.activemq.artemis.core.server.transformer.Transformer`
  interface.

  If specified, then the transformer's `transform()` method will be invoked with the message before
  it is transferred. This gives you the opportunity to transform the message's header or body
  before it is federated.

- **property**

  Holds key-value pairs that can be used to configure the transformer.

---

**Upstream parameters**

Tag *upstream* defines the upstream broker connection and the policies to use.

- **name**

  Unique name for upstream federated server.

- **user**

  This optional attribute determines the username to use when creating the upstream connection
  to the remote server. If not specified, the shared federation user and password will be used if
  they were set.

- **password**

  This optional attribute determines the password to use when creating the upstream connection
  to the remote server. If not specified, the shared federation user and password will be used if
  they were set.

- **static-connectors**

  Either this or *discovery-group-ref* is used to connect the bridge to the target server.

  The *static-connectors* is a list of *connector-ref* elements pointing to *connector* elements defined
elsewhere. A *connector* encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc.) as well as the server connection parameters (host, port etc).

For more information about what connectors are and how to configure them, see Configuring the Transport.

**discovery-group-ref**

Either this or *static-connectors* is used to connect the bridge to the target server.

The discovery-group-ref element has one attribute - discovery-group-name. This attribute points to a discovery-group defined elsewhere. For more information about what discovery-groups are and how to configure them, see Discovery Groups.

**ha**

This optional parameter determines whether this bridge should support high availability. True means it will connect to any available server in a cluster and support failover. Default is false.

**circuit-breaker-timeout**

When a connection issue occurs, as the single connection is shared by many federated queue and address consumers, to avoid each one trying to reconnect and possibly causing a thundering herd issue, the first one will be tried. If unsuccessful the circuit breaker will open, returning the same exception to all connections. This is the timeout until the circuit can be closed and connection retried. Measured in milliseconds.

**share-connection**

If there is a downstream and upstream connection configured for the same broker then the same connection will be shared as long as both stream configs set this flag to true. Default is false.

**check-period**

The period (in milliseconds) used to check if the federation connection has failed to receive pings from another server. Default is 30000.

**connection-ttl**

This is how long a federation connection should stay alive if it stops receiving messages from the remote broker. Default is 60000.

**call-timeout**

When a packet is sent via a federation connection and is a blocking call, i.e. for acknowledgements, this is how long it will wait (in milliseconds) for the reply before throwing an exception. Default is 30000.

**call-failover-timeout**

Similar to call-timeout but used when a call is made during a failover attempt. Default is -1 (no timeout).

**retry-interval**

This optional parameter determines the period in milliseconds between subsequent
reconnection attempts, if the connection to the target server has failed. The default value is 500 milliseconds.

**retry-interval-multiplier**

Used to increase the retry-interval after each reconnect attempt, default is 1.

**max-retry-interval**

The maximum delay (in milliseconds) for retries. Default is 2000.

**initial-connect-attempts**

The number of times the system will try to connect to the remote broker in the federation. If the max-retry is achieved this broker will be considered permanently down and the system will not route messages to this broker. Default is -1 (infinite retries).

**reconnect-attempts**

The number of times the system will try to reconnect to the remote broker in the federation. If the max-retry is achieved this broker will be considered permanently down and the system will stop routing messages to this broker. Default is -1 (infinite retries).

### 35.3. Configuring Downstream Federation

Similarly to upstream configuration, a downstream configuration can be configured. This works by sending a command to the downstream broker to have it create an upstream connection back to the downstream broker. The benefit of this is being able to configure everything for federation on one broker in some cases to make it easier, such as a hub and spoke topology.

All the same configuration options apply to downstream as does upstream with the exception of one extra configuration flag that needs to be set:

**upstream-connector-ref**

Is an element pointing to a connector elements defined elsewhere. This reference is used to tell the downstream broker what connector to use to create a new upstream connection back to the downstream broker.

A connector encapsulates knowledge of what transport to use (TCP, SSL, HTTP etc.) as well as the server connection parameters (host, port etc). For more information about what connectors are and how to configure them, see Configuring the Transport.

Sample Downstream Address Federation setup:

```xml
<!--Other config Here -->

<connectors>
  <connector name="netty-connector">tcp://localhost:61616</connector>
  <connector name="eu-west-1-connector">tcp://localhost:61616</connector>
  <connector name="eu-east-1-connector">tcp://localhost:61617</connector>
</connectors>
```
<acceptors>
  <acceptor name="netty-acceptor">tcp://localhost:61616</acceptor>
</acceptors>

<!--Other config Here -->

<federations>
  <federation name="eu-north-1" user="federation_username"
              password="32a10275cf4ab4e9">
    <downstream name="eu-east-1">
      <static-connectors>
        <connector-ref>eu-east-1-connector</connector-ref>
      </static-connectors>
      <upstream-connector-ref>netty-connector</upstream-connector-ref>
      <policy ref="news-address-federation"/>
    </downstream>
    <downstream name="eu-west-1">
      <static-connectors>
        <connector-ref>eu-west-1-connector</connector-ref>
      </static-connectors>
      <upstream-connector-ref>netty-connector</upstream-connector-ref>
      <policy ref="news-address-federation"/>
    </downstream>
  </federation>
  <queue-policy name="news-queue-federation" priority-adjustment="-5"
                include-federated="true" transformer-ref="news-transformer">
    <include queue-match="#" address-match="queue.bbc.new"/>
    <include queue-match="#" address-match="queue.usatoday"/>
    <include queue-match="#" address-match="queue.news.#"/>
    <exclude queue-match="#.local" address-match="#"/>
  </queue-policy>
  <transformer name="news-transformer">
    <class-name>org.foo.NewsTransformer</class-name>
    <property key="key1" value="value1"/>
    <property key="key2" value="value2"/>
  </transformer>
</federations>
Chapter 36. High Availability and Failover

We define high availability (HA) as the *ability for the system to continue functioning after failure of one or more of the servers.*

A part of high availability is *failover* which we define as the *ability for client connections to migrate from one server to another in event of server failure so client applications can continue to operate.*

### 36.1. Terminology

In order to discuss both configuration and runtime behavior consistently we need to define a couple nouns and adjectives. These terms will be used throughout the documentation, configuration, source code, and runtime logs.

#### 36.1.1. Configuration

These nouns identify how the broker is *configured*, e.g. in `broker.xml`. The configuration allows brokers to be paired together as a primary/backup group (i.e. an HA pair of brokers).

**primary**

This identifies the main broker in the high availability configuration. Oftentimes the hardware on this broker will be higher performance than the hardware on the backup broker. Typically, this broker is started before the backup and is active most of the time. Each primary server can have 1 or more backup servers. However, only one backup will take over the primary server's work.

**backup**

This identifies the broker that should take over when the primary broker fails in a high availability configuration. Oftentimes the hardware on this broker will be lower performance than the hardware on the primary broker. Typically, this broker is started after the primary and is passive most of the time.

#### 36.1.2. Runtime

These adjectives describe the *behavior* of the broker at runtime. For example, you could have a passive primary or an active backup.

**active**

This identifies a broker in a high-availability configuration which is accepting remote connections. For example, consider the scenario where the primary broker has failed and its backup has taken over. The backup would be described as active at that point since it is accepting remote connections.

**passive**

This identifies a broker in a high-availability configuration which is not accepting remote connections. For example, consider the scenario where the primary broker was started and then the backup broker was started. The backup broker would be passive since it is not accepting
remote connections. It is waiting for the primary to fail before it activates and begins accepting remote connections.

36.2. HA Policies

Apache ActiveMQ Artemis supports two main policies for backing up a server:

- shared store
- replication

These are configured via the **ha-policy** configuration element.

**What is Backed Up?**

Only message data **written to storage** will survive failover. Any message data not written to storage will not be available after failover.

**Clustering is Required**

A proper **cluster** configuration is required as a pre-requisite for an HA configuration. The cluster configuration allows the server to announce its presence to its primary/backup (or any other nodes in the cluster).

There is technically a third policy called **primary-only** which omits the backup entirely. This is used to configure **scale-down**. This is the default policy if none is provided.

36.2.1. Shared Store

When using a shared store both primary and backup servers share the **same** entire data directory using a shared file system. This includes the paging directory, journal directory, large messages, and bindings journal.

When the primary server fails it will release its lock on the shared journal and allow the backup server to activate. The backup will then load the data from the shared file system and accept remote connections from clients.

Typically, this will be some kind of high performance Storage Area Network (SAN). Network Attached Storage (NAS), like an **NFS mount**, is viable but won’t provide optimal performance.

One main advantage of a shared store configuration is that no replication occurs between the primary and backup nodes which means it does not suffer any performance penalties due to the overhead of replication during normal operation.

One potentially significant disadvantage of shared store versus replication is that it requires a shared file system, and when the backup server activates it needs to load the journal from the shared store which can take some time depending on the amount of data in the store and the speed of the store.

If you require the highest performance during normal operation then acquire access to a fast SAN and deal with a slightly slower failover (depending on amount of data).
What About Split Brain?

Shared store configurations are naturally immune to split-brain.

Shared Store Configuration

Both primary & backup servers must configure the location of journal directories to the same shared location (as explained in persistence documentation).

Primary

The primary broker needs this basic configuration in broker.xml:

```xml
<ha-policy>
  <shared-store>
    <primary/>
  </shared-store>
</ha-policy>
```

Additional parameters

failover-on-shutdown

Whether the graceful shutdown of this primary broker will cause the backup to activate. Default is false which means that only a broker crash or forceful shutdown (e.g. using ctrl-c) will trigger the backup to activate.

wait-for-activation

This setting is only for embedded use cases where the primary broker has failed, the backup has activated, and the primary has been restarted. By default, when org.apache.activemq.artemis.core.server.ActiveMQServer.start() is invoked the broker will block until the primary broker actually takes over from the backup (i.e. either via failback or by the backup actually stopping). Setting wait-for-activation to false prevents start() from blocking so that control is returned to the caller. The caller can use waitForActivation() to wait until broker activates or just check the current status using getState(). Default is true.

Backup

The backup needs this basic configuration in broker.xml:

```xml
<ha-policy>
  <shared-store>
    <backup/>
  </shared-store>
</ha-policy>
```

Additional parameters
allow-failback
Whether this backup will automatically stop when its primary is restarted and requests to take over its place. The use case is when a primary server stops and its backup takes over its duties, later the primary server restarts and requests the now-active backup to stop so the primary can take over again. Default is true.

failover-on-shutdown
Whether the graceful shutdown of this primary broker will cause the backup to activate. Default is false which means that only a broker crash or forceful shutdown (e.g. using ctrl-c) will trigger the backup to activate. This only applies when this backup has activated due to its primary failing.

scale-down
If provided then this backup will scale down rather than becoming active after fail over. This really only applies to colocated configurations where the backup will scale-down its messages to the primary broker in the same JVM.

restart-backup
Will this backup restart after being stopped due to failback or scaling down. Default is false.

NFS Mount Recommendations
If you choose to implement your shared store configuration with NFS here are some recommended configuration options. These settings are designed for reliability and to help the broker detect problems with NFS quickly and shut itself down so that clients can failover to a working broker.

sync
Specifies that all changes are immediately flushed to disk.

intr
Allows NFS requests to be interrupted if the server is shut down or cannot be reached.

noac
Disables attribute caching. This behavior is needed to achieve attribute cache coherence among multiple clients.

soft
Specifies that if the NFS server is unavailable the error should be reported rather than waiting for the server to come back online.

lookupcache=none
Disables lookup caching.

timeo=n
The time, in deciseconds (i.e. tenths of a second), that the NFS client (i.e. the broker) waits for a response from the NFS server before it retries a request. For NFS over TCP the default timeo value is 600 (60 seconds). For NFS over UDP the client uses an adaptive algorithm to estimate an appropriate timeout value for frequently used request types, such as read and write requests.
retrans=n
The number of times that the NFS client retries a request before it attempts further recovery action.

Use reasonable values when you configure timeo and retrans. A default timeo wait time of 600 deciseconds (60 seconds) combined with a retrans value of 5 retries can result in a five-minute wait for the broker to detect an NFS disconnection. You likely don’t want all store-related operations on the broker to be blocked for that long while clients wait for responses. Tune these values to balance latency and reliability in your environment.

36.2.2. Replication

When using replication, the primary and the backup servers do not share the same data directories. All data synchronization is done over the network. Therefore, all (durable) data received by the primary server will be duplicated to the backup.

Note that upon start-up the backup server will first need to synchronize all existing data from the primary server before becoming capable of replacing the primary server should it fail. Therefore, unlike when using shared storage, a backup will not be fully operational until after it finishes synchronizing the data with its primary server. The time it takes for this to happen depends on the amount of data to be synchronized and the connection speed.

In general, synchronization occurs in parallel with current network traffic so this won’t cause any blocking for current clients. However, there is a critical moment at the end of this process where the replicating server must complete the synchronization and ensure the backup acknowledges this completion. This exchange between the replicating server and backup will block any journal related operations. The maximum length of time that this exchange will block is controlled by the initial-replication-sync-timeout configuration element.

Since replication will create a copy of the data at the backup then in case of a successful fail-over, the backup’s data will be newer than the primary’s data. If you configure your backup to allow failback to the primary then when the primary is restarted it will be passive and the active backup will synchronize its data with the passive primary before stopping to allow the passive primary to become active again. If both servers are shutdown then the administrator will have to determine which one has the latest data.

An Important Difference From Shared Store
If a shared-store backup does not find a primary then it will just activate and service client requests like it is a primary.

However, in the replication case, the backup just keeps waiting for a primary to pair with because the backup does not know whether its data is up-to-date. It cannot unilaterally decide to activate. To activate a replicating backup using its current data the administrator must change its configuration to make it a primary server by changing backup to primary.
Split Brain

"Split Brain" is a potential issue that is important to understand. A whole chapter has been devoted to explaining what it is and how it can be mitigated at a high level. Once you read it you will understand the main differences between quorum voting and pluggable lock manager configurations which will be referenced in later sections.

Replication Configuration

In a shared-store configuration brokers pair with each other based on their shared storage device. However, since replication configurations have no such shared storage device they must find each other another way. Servers can be grouped together explicitly using the same group-name in both the primary or the backup elements. A backup will only connect to a primary that shares the same node group name.

A group-name Example

Suppose you have 5 primary servers and 6 backup servers:

- primary1, primary2, primary3: with group-name=fish
- primary4, primary5: with group-name=bird
- backup1, backup2, backup3, backup4: with group-name=fish
- backup5, backup6: with group-name=bird

After joining the cluster the backups with group-name=fish will search for primary servers with group-name=fish to pair with. Since there is one backup too many, the fish will remain with one spare backup.

The 2 backups with group-name=bird (backup5 and backup6) will pair with primary servers primary4 and primary5.

If group-name is not configured then the backup will search for any primary that it can find the cluster. It tries to replicate with each primary until it finds a primary that has no current backup configured. If no primary server is available it will wait until the cluster topology changes and repeat the process.

Primary

The primary broker needs this basic configuration in broker.xml:

```xml
<ha-policy>
  <replication>
    <primary/>
  </replication>
</ha-policy>
```
Additional parameters

**group-name**

If set, backup servers will only pair with primary servers with matching group-name. See above for more details. Valid for both quorum voting and pluggable lock manager.

**cluster-name**

Name of the `cluster-connection` to use for replication. This setting is only necessary if you configure multiple cluster connections. If configured then the connector configuration of the cluster configuration with this name will be used when connecting to the cluster to discover if an active server is already running, see `check-for-active-server`. If unset then the default cluster connections configuration is used (i.e. the first one configured). Valid for both quorum voting and pluggable lock manager.

**max-saved-replicated-journals-size**

This option specifies how many replication backup directories will be kept when server starts as a passive backup. Every time when server starts as such all former data moves to `oldreplica.{id}` directory, where `{id}` is a growing backup index. This parameter sets the maximum number of such directories kept on disk. Valid for both quorum voting and pluggable lock manager.

**check-for-active-server**

Whether to check the cluster for an active server using our own server ID when starting up. This is an important option to avoid split-brain when failover happens and the primary is restarted. Default is `false`. Only valid for quorum voting.

**initial-replication-sync-timeout**

The amount of time the replicating server will wait at the completion of the initial replication process for the backup to acknowledge it has received all the necessary data. The default is `30000`; measured in milliseconds. Valid for both quorum voting and pluggable lock manager.

During this interval any journal related operations will be blocked.

**vote-on-replication-failure**

Whether this primary broker should vote to remain active if replication is lost. Default is `false`. Only valid for quorum voting.

**quorum-size**

The quorum size used for voting after replication loss, -1 means use the current cluster size Default is `-1`. Only valid for quorum voting.

**vote-retries**

If we start as a backup and lose connection to the primary, how many times should we attempt to vote for quorum before restarting Default is `12`. Only valid for quorum voting.

**vote-retry-wait**

How long to wait (in milliseconds) between each vote attempt. Default is `5000`. Only valid for quorum voting.
quorum-vote-wait

How long to wait (in seconds) for vote results. Default is 30. Only valid for quorum voting.

retry-replication-wait

If we start as a backup how long to wait (in milliseconds) before trying to replicate again after failing to find a primary. Default is 2000. Valid for both quorum voting and pluggable lock manager.

manager

This element controls and is required for pluggable lock manager configuration. It has two sub-elements:

- **class-name** - the name of the class implementing org.apache.activemq.artemis.lockmanager.DistributedLockManager. Default is org.apache.activemq.artemis.lockmanager.zookeeper.CuratorDistributedLockManager which integrates with ZooKeeper.

- **properties** - a list of property elements each with key and value attributes for configuring the plugin.

Here's a simple example:

```xml
<ha-policy>
  <replication>
    <primary>
      <manager>
        <class-name>org.foo.MyQuorumVotingPlugin</class-name>
        <properties>
          <property key="property1" value="value1"/>
          <property key="property2" value="value2"/>
        </properties>
      </manager>
    </primary>
  </replication>
</ha-policy>
```

coordination-id

This is for Competing Primary Brokers. Only valid when using pluggable lock manager.

Backup

The backup needs this basic configuration in broker.xml:

```xml
<ha-policy>
  <replication>
    <backup/>
  </replication>
</ha-policy>
```
Additional parameters

**group-name**
If set, backup servers will only pair with primary servers with matching group-name. See above for more details. Valid for both quorum voting and pluggable lock manager.

**cluster-name**
Name of the `cluster-connection` to use for replication. This setting is only necessary if you configure multiple cluster connections. If configured then the connector configuration of the cluster configuration with this name will be used when connecting to the cluster to discover if an active server is already running, see `check-for-active-server`. If unset then the default cluster connections configuration is used (i.e. the first one configured). Valid for both quorum voting and pluggable lock manager.

**max-saved-replicated-journals-size**
This option specifies how many replication backup directories will be kept when server starts as a passive backup. Every time when server starts as such all former data moves to `oldreplica.{id}` directory, where `{id}` is a growing backup index. This parameter sets the maximum number of such directories kept on disk. Valid for both quorum voting and pluggable lock manager.

**scale-down**
If provided then this backup will scale down rather than becoming active after fail over. This really only applies to colocated configurations where the backup will scale-down its messages to the primary broker in the same JVM.

**restart-backup**
Will this server, if a backup, restart once it has been stopped because of failback or scaling down. Default is `false`.

**allow-failback**
Whether this backup will automatically stop when its primary is restarted and requests to take over its place. The use case is when a primary server stops and its backup takes over its duties, later the primary server restarts and requests the now-active backup to stop so the primary can take over again. Default is `true`. Valid for both quorum voting and pluggable lock manager.

**initial-replication-sync-timeout**
After failover when the backup has activated this is enforced when the primary is restarted and connects as a backup (e.g. for failback). The amount of time the replicating server will wait at the completion of the initial replication process for the backup to acknowledge it has received all the necessary data. The default is `30000`; measured in milliseconds. Valid for both quorum voting and pluggable lock manager.

During this interval any journal related operations will be blocked.

**vote-on-replication-failure**
Whether this primary broker should vote to remain active if replication is lost. Default is `false`. Only valid for quorum voting.
**quorum-size**

The quorum size used for voting after replication loss, -1 means use the current cluster size. Default is **-1**. Only valid for quorum voting.

**vote-retries**

If we start as a backup and lose connection to the primary, how many times should we attempt to vote for quorum before restarting. Default is **12**. Only valid for quorum voting.

**vote-retry-wait**

How long to wait (in milliseconds) between each vote attempt. Default is **5000**. Only valid for quorum voting.

**quorum-vote-wait**

How long to wait (in seconds) for vote results. Default is **30**. Only valid for quorum voting.

**retry-replication-wait**

If we start as a backup how long to wait (in milliseconds) before trying to replicate again after failing to find a primary. Default is **2000**. Valid for both quorum voting and pluggable lock manager.

**manager**

This element controls and is required for pluggable lock manager configuration. It has two sub-elements:

- **class-name** - the name of the class implementing `org.apache.activemq.artemis.lockmanager.DistributedLockManager`. Default is `org.apache.activemq.artemis.lockmanager.zookeeper.CuratorDistributedLockManager` which integrates with ZooKeeper.

- **properties** - a list of property elements each with key and value attributes for configuring the plugin.

Here’s a simple example:

```xml
<ha-policy>
  <replication>
    <backup>
      <manager>
        <class-name>org.foo.MyQuorumVotingPlugin</class-name>
        <properties>
          <property key="property1" value="value1"/>
          <property key="property2" value="value2"/>
        </properties>
      </manager>
      <allow-failback>true</allow-failback>
    </backup>
  </replication>
</ha-policy>
```
Apache ZooKeeper Integration

The default pluggable lock manager implementation uses Apache Curator to integrate with Apache ZooKeeper.

ZooKeeper Plugin Configuration

Here's a basic configuration example:

```xml
<ha-policy>
  <replication>
    <primary>
      <manager>
        <class-name>org.apache.activemq.artemis.lockmanager.zookeeper.CuratorDistributedLockManager</class-name>
        <properties>
          <property key="connect-string" value="127.0.0.1:6666,127.0.0.1:6667,127.0.0.1:6668"/>
        </properties>
      </manager>
    </primary>
  </replication>
</ha-policy>
```

+ NOTE: The `class-name` isn't technically required here since the default value is being used, but it is included for clarity.

Available Properties

- **connect-string**
  (no default)

- **session-ms**
  (default is 18000 ms)

- **session-percent**
  (default is 33); should be \( \leq \) default (see [TN14](https://cwiki.apache.org/confluence/display/CURATOR/TN14) for more info)

- **connection-ms**
  (default is 8000 ms)

- **retries**
  (default is 1)

- **retries-ms**
  (default is 1000 ms)
Improving Reliability

Configuration of the ZooKeeper ensemble is the responsibility of the user, but here are few suggestions to improve the reliability of the quorum service:

- Broker \(<\text{code}>\text{session\_ms}\</\text{code}>\) must be \(<\text{code}>\geq 2 * \text{server tick time}\</\text{code}>\) and \(<\text{code}>\leq 20 * \text{server tick time}\</\text{code}>\) as by \(<\text{a href="https://zookeeper.apache.org/doc/r3.6.3/zookeeperAdmin.html"}>\text{ZooKeeper 3.6.3 admin guide}\</a>\). This directly impacts how fast a backup can failover to an isolated/killed/unresponsive primary. The higher, the slower.

- GC on broker machine should allow keeping GC pauses within 1/3 of \(<\text{code}>\text{session\_ms}\</\text{code}>\) in order to let the ZooKeeper heartbeat protocol work reliably. If that is not possible, it is better to increase \(<\text{code}>\text{session\_ms}\</\text{code}>\), accepting a slower failover.

- ZooKeeper must have enough resources to keep GC (and OS) pauses much smaller than server tick time. Please consider carefully if a broker and ZooKeeper node should share the same physical machine depending on the expected load of the broker.

- Network isolation protection requires configuring \(<\text{code}>\geq 3\</\text{code}>\) ZooKeeper nodes

As noted previously, \(<\text{code}>\text{session\_ms}\</\text{code}>\) affects the failover duration. The passive broker can activate after \(<\text{code}>\text{session\_ms}\</\text{code}>\) expires or if the active broker voluntary gives up its role (e.g. during a failback/manual broker stop, it happens immediately).

For the former case (session expiration with active broker no longer present), the passive broker can detect an unresponsive active broker by using:

1. cluster connection PINGs (affected by \(<\text{code}>\text{connection-ttl}\</\text{code}>\) tuning)
2. closed TCP connection notification (depends on TCP configuration and networking stack/topology)

The suggestion is to tune \(<\text{code}>\text{connection-ttl}\</\text{code}>\) low enough to attempt failover as soon as possible, while taking in consideration that the whole fail-over duration cannot last less than the configured \(<\text{code}>\text{session\_ms}\</\text{code}>\).

A backup still needs to carefully configure \(<\text{code}>\text{connection-ttl}\</\text{code}>\) in order to promptly send a request to the quorum manager to become active before failing-over.

Competing Primary Brokers

When delegating quorum to pluggable implementation roles of primary & backup are less important. It is possible to have two brokers \(<\text{code}>\text{compete}\</\text{code}>\) for activation with the winner activating as primary and the loser taking the backup role. On restart, any peer server with the most up-to-date journal can activate. The key is that the brokers need to know in advance what identity they will coordinate on. In the replication \(<\text{code}>\text{primary ha-policy}\</\text{code}>\) we can explicitly set the \(<\text{code}>\text{coordination-id}\</\text{code}>\) to a common value for all peers in a cluster.
36.3. Failing Back to Primary Server

After a primary server has failed and a backup taken has taken over its duties, you may want to restart the primary server and have clients fail back.

36.3.1. Failback with Shared Store

In case of shared storage you have a couple of options:

1. Simply restart the primary and kill the backup. You can do this by killing the process itself.

2. Alternatively you can set `allow-failback` to `true` on the backup which will force the backup that has become active to automatically stop. This configuration would look like:

   ```xml
   <ha-policy>
   <shared-store>
     <backup>
       <allow-failback>true</allow-failback>
     </backup>
   </shared-store>
   </ha-policy>
   ```

It is also possible, in the case of shared store, to cause failover to occur on normal server shutdown, to enable this set the following property to true in the `ha-policy` configuration on either the `primary` or `backup` like so:
By default this is set to false, if by some chance you have set this to false but still want to stop the server normally and cause failover then you can do this by using the management API as explained at Management.

You can also force the active backup to shutdown when the primary comes back up allowing the primary to take over automatically by setting the following property in the broker.xml configuration file as follows:

```xml
<ha-policy>
  <shared-store>
    <backup>
      <allow-failback>true</allow-failback>
    </backup>
  </shared-store>
</ha-policy>
```

36.3.2. Failback with Replication

As with shared storage the allow-failback option can be set for both quorum voting and pluggable lock manager replication configurations.

Quorum Voting

```xml
<ha-policy>
  <replication>
    <backup>
      <allow-failback>true</allow-failback>
    </backup>
  </replication>
</ha-policy>
```

With quorum voting replication you need to set an extra property check-for-active-server to true in the primary configuration. If set to true then during start-up the primary server will first search the cluster for another active server using its nodeID. If it finds one it will contact this server and try to “failback”. Since this is a remote replication scenario the primary will have to synchronize its data with the backup server running with its ID. Once they are in sync it will request the other server (which it assumes it is a backup that has assumed its duties) to shutdown in order for it to take over. This is necessary because otherwise the primary server has no means to know whether
there was a fail-over or not, and if there was, if the server that took its duties is still running or not.
To configure this option at your `broker.xml` configuration file as follows:

```xml
<ha-policy>
  <replication>
    <primary>
      <check-for-active-server>true</check-for-active-server>
    </primary>
  </replication>
</ha-policy>
```

Be aware that if you restart a primary server after failover has occurred then `check-for-active-server` must be `true`. If not then the primary server will restart and serve the same messages that the backup has already handled causing duplicates.

**Pluggable Lock Manager**

One key difference between replication with quorum voting and replication with a lock manager is that with quorum voting if the primary cannot reach any active server with its nodeID then it activates unilaterally. With a pluggable lock manager the responsibilities of coordination are delegated to 3rd party. There are no unilateral decisions. The primary will only activate when it knows that it has the most up-to-date version of the journal identified by its nodeID.

In short: **a primary cannot activate without permission when using a pluggable lock manager.**

Here's an example configuration:

```xml
<ha-policy>
  <replication>
    <manager>
      <!-- some meaningful configuration -->
    </manager>
    <primary>
      <!-- no need to check-for-active-server anymore -->
    </primary>
  </replication>
</ha-policy>
```

### 36.3.3. All Shared Store Configuration

**Primary**

The following lists all the `ha-policy` configuration elements for HA strategy shared store for `primary`:

**failover-on-shutdown**

If set to `true` then when this server is stopped normally the backup will become active assuming
failover. If false then the backup server will remain passive. Note that if false and you want failover to occur then you can use the management API as explained at Management.

**wait-for-activation**
If set to true then server startup will wait until it is activated. If set to false then server startup will be done in the background. Default is true.

**Backup**
The following lists all the ha-policy configuration elements for HA strategy Shared Store for backup:

**failover-on-shutdown**
In the case of a backup that has become active then when set to true and this server is stopped normally the passive primary will become active assuming failover. If false then the primary server will remain passive. Note that if false and you want failover to occur then you can use the management API as explained at Management.

**allow-failback**
Whether a server will automatically stop when another places a request to take over its place. The use case is when the backup has failed over.

**Colocated Backup Servers**
It is also possible when running standalone to colocate backup servers in the same JVM as another primary server. Primary Servers can be configured to request another primary server in the cluster to start a backup server in the same JVM either using shared store or replication. The new backup server will inherit its configuration from the primary server creating it apart from its name, which will be set to colocated_backup_n where n is the number of backups the server has created, and any directories and its Connectors and Acceptors which are discussed later on in this chapter. A primary server can also be configured to allow requests from backups and also how many backups a primary server can start. This way you can evenly distribute backups around the cluster. This is configured via the ha-policy element in the broker.xml file like so:

```xml
<ha-policy>
  <replication>
    <colocated>
      <request-backup>true</request-backup>
      <max-backups>1</max-backups>
      <backup-request-retries>-1</backup-request-retries>
      <backup-request-retry-interval>5000</backup-request-retry-interval>
      <primary/>
      <backup/>
    </colocated>
  </replication>
</ha-policy>
```

the above example is configured to use replication, in this case the primary and backup configurations must match those for normal replication as in the previous chapter. shared-store is
also supported

36.3.4. Configuring Connectors and Acceptors

If the HA Policy is **colocated** then **connectors** and **acceptors** will be inherited from the primary server creating it and offset depending on the setting of **backup-port-offset** configuration element. If this is set to say 100 (which is the default) and a connector is using port 61616 then this will be set to 61716 for the first server created, 61816 for the second, and so on.

![Diagram of HA setup](image)

For INVM connectors and Acceptors the id will have **colocated_backup_n** appended, where n is the backup server number.

36.3.5. Remote Connectors

It may be that some of the Connectors configured are for external servers and hence should be excluded from the offset. For instance a connector used by the cluster connection to do quorum voting for a replicated backup server, these can be omitted from being offset by adding them to the **ha-policy** configuration like so:

```xml
<ha-policy>
    <replication>
        <colocated>
            ...
            <excludes>
                <connector-ref>remote-connector</connector-ref>
            </excludes>
            ...
        </colocated>
    </replication>
</ha-policy>
```

36.3.6. Configuring Directories

Directories for the Journal, Large messages and Paging will be set according to what the HA strategy is. If shared store the requesting server will notify the target server of which directories to
use. If replication is configured then directories will be inherited from the creating server but have the new backups name appended.

The following table lists all the `ha-policy` configuration elements for colocated policy:

**request-backup**
If true then the server will request a backup on another node

**backup-request-retries**
How many times the primary server will try to request a backup, `-1` means for ever.

**backup-request-retry-interval**
How long to wait for retries between attempts to request a backup server.

**max-backups**
How many backups a primary server can create

**backup-port-offset**
The offset to use for the Connectors and Acceptors when creating a new backup server.

### 36.4. Scaling Down

An alternative to using primary/backup groups is to configure `scaledown`. When configured for scale down a server can copy all its messages and transaction state to another active server. The advantage of this is that you don’t need full backups to provide some form of HA, however there are disadvantages with this approach the first being that it only deals with a server being stopped and not a server crash. The caveat here is if you configure a backup to scale down.

Another disadvantage is that it is possible to lose message ordering. This happens in the following scenario, say you have 2 active servers and messages are distributed evenly between the servers from a single producer, if one of the servers scales down then the messages sent back to the other server will be in the queue after the ones already there, so server 1 could have messages 1,3,5,7,9 and server 2 would have 2,4,6,8,10, if server 2 scales down the order in server 1 would be 1,3,5,7,9,2,4,6,8,10.

The configuration for an active server to scale down would be something like:
In this instance the server is configured to use a specific connector to scale down, if a connector is not specified then the first INVM connector is chosen, this is to make scale down from a backup server easy to configure. It is also possible to use discovery to scale down, this would look like:

```xml
<ha-policy>
  <primary-only>
    <scale-down>
      <connectors>
        <connector-ref>server1-connector</connector-ref>
      </connectors>
    </scale-down>
  </primary-only>
</ha-policy>
```

36.4.1. Scale Down with groups

It is also possible to configure servers to only scale down to servers that belong in the same group. This is done by configuring the group like so:

```xml
<ha-policy>
  <primary-only>
    <scale-down>
      ...
      <group-name>my-group</group-name>
    </scale-down>
  </primary-only>
</ha-policy>
```

In this scenario only servers that belong to the group `my-group` will be scaled down to

36.4.2. Scale Down and Backups

It is also possible to mix scale down with HA via backup servers. If a backup is configured to scale down then after failover has occurred, instead of starting fully the backup server will immediately scale down to another active server. The most appropriate configuration for this is using the colocated approach. It means that as you bring up primary servers they will automatically be backed up, and as they are shutdown their messages are made available on another active server. A typical configuration would look like:
36.4.3. Scale Down and Clients

When a server is stopping and preparing to scale down it will send a message to all its clients informing them which server it is scaling down to before disconnecting them. At this point the client will reconnect however this will only succeed once the server has completed the scaledown process. This is to ensure that any state such as queues or transactions are there for the client when it reconnects. The normal reconnect settings apply when the client is reconnecting so these should be high enough to deal with the time needed to scale down.

36.5. Client Failover

Apache ActiveMQ Artemis clients can be configured to receive knowledge of all primary and backup servers, so that in event of connection failure the client will detect this and reconnect to the backup server. The backup server will then automatically recreate any sessions and consumers that existed on each connection before failover, thus saving the user from having to hand-code manual reconnection logic. For further details see Client Failover
A Note on Seamless Failover

Apache ActiveMQ Artemis does not reproduce full server state between active and passive servers. When a core client automatically creates a new session on the backup that session won't contain any information about messages already sent or acknowledged in the previous session. Any in-flight sends or acknowledgements at the time of failover will also be lost if they weren't written to storage.

Theoretically we could provide a 100% transparent, seamless failover which would avoid any lost messages or acknowledgements. However, this comes at a great cost: reproducing the full server state (including the queues, session, etc.). This would require every operation on the primary server to be reproduced on the backup server in the exact same global order to ensure a consistent state. This is extremely hard to do in a performant and scalable way, especially when one considers that multiple threads are changing the active server's state concurrently.

It is possible to provide reproduce the full state machine using techniques such as virtual synchrony, but this does not scale well and effectively serializes all operations to a single thread, dramatically reducing concurrency.

Other techniques for multi-threaded use-cases exist such as reproducing lock states or thread scheduling, but this is very hard to achieve at a Java level.

Consequently, it has been decided that it worth not worth massively reducing performance and concurrency for the sake of 100% transparent failover. Even without 100% transparent failover, it is simple to guarantee once and only once delivery, even in the case of failure, by using a combination of duplicate detection and retrying of transactions. However, this is not 100% transparent to the client code.

36.5.1. Handling Blocking Calls During Failover

If the client code is in a blocking call to the server, waiting for a response to continue its execution, when failover occurs, the new session will not have any knowledge of the call that was in progress. This call might otherwise hang for ever, waiting for a response that will never come.

To prevent this, Apache ActiveMQ Artemis will unblock any blocking calls that were in progress at the time of failover by making them throw a javax.jms.JMSException (if using JMS), or a ActiveMQException with error code ActiveMQException.UNBLOCKED. It is up to the client code to catch this exception and retry any operations if desired.

If the method being unblocked is a call to commit(), or prepare(), then the transaction will be automatically rolled back and Apache ActiveMQ Artemis will throw a javax.jms.TransactionRolledBackException (if using JMS), or a ActiveMQException with error code ActiveMQException.TRANSACTION_ROLLED_BACK if using the core API.

36.5.2. Handling Failover With Transactions

If the session is transactional and messages have already been sent or acknowledged in the current
transaction, then the server cannot be sure that messages sent or acknowledgements have not been lost during the failover.

Consequently the transaction will be marked as rollback-only, and any subsequent attempt to commit it will throw a `javax.jms.TransactionRolledBackException` (if using JMS), or an `ActiveMQException` with error code `ActiveMQException.TRANSACTION_ROLLED_BACK` if using the core API.

The caveat to this rule is when XA is used either via JMS or through the core API. If 2 phase commit is used and prepare has already been called then rolling back could cause a `HeuristicMixedException`. Because of this the commit will throw a `XAException.XA_RETRY` exception. This informs the Transaction Manager that it should retry the commit at some later point in time, a side effect of this is that any non persistent messages will be lost. To avoid this use persistent messages when using XA. With acknowledgements this is not an issue since they are flushed to the server before prepare gets called.

It is up to the user to catch the exception, and perform any client side local rollback code as necessary. There is no need to manually rollback the session - it is already rolled back. The user can then just retry the transactional operations again on the same session.

Apache ActiveMQ Artemis ships with a fully functioning example demonstrating how to do this, please see the examples chapter.

If failover occurs when a commit call is being executed, the server, as previously described, will unblock the call to prevent a hang, since no response will come back. In this case it is not easy for the client to determine whether the transaction commit was actually processed before failure occurred.

If XA is being used either via JMS or through the core API then an `XAException.XA_RETRY` is thrown. This is to inform Transaction Managers that a retry should occur at some point. At some later point in time the Transaction Manager will retry the commit. If the original commit has not occurred then it will still exist and be committed, if it does not exist then it is assumed to have been committed although the transaction manager may log a warning.

To remedy this, the client can simply enable duplicate detection (Duplicate Message Detection) in the transaction, and retry the transaction operations again after the call is unblocked. If the transaction had indeed been committed successfully before failover, then when the transaction is retried, duplicate detection will ensure that any durable messages resent in the transaction will be ignored on the server to prevent them getting sent more than once.

By catching the rollback exceptions and retrying, catching unblocked calls and enabling duplicate detection, *once and only once* delivery guarantees can be provided for messages in the case of failure, guaranteeing 100% no loss or duplication of messages.
Handling Failover With Non Transactional Sessions

If the session is non transactional, messages or acknowledgements can be lost in the event of a failover.

If you wish to provide *once and only once* delivery guarantees for non transacted sessions too, enable duplicate detection, and catch unblock exceptions as described in [Handling Blocking Calls During Failover](#).

**Use client connectors to fail over**

Apache ActiveMQ Artemis clients retrieve the backup connector from the topology updates that the cluster brokers send. If the connection options of the clients don’t match the options of the cluster brokers the clients can define a client connector that will be used in place of the connector in the topology. To define a client connector it must have a name that matches the name of the connector defined in the `cluster-connection` of the broker, i.e. supposing to have a primary broker with the cluster connector name `node-0` and a backup broker with the `cluster-connector` name `node-1` the client connection url must define 2 connectors with the names `node-0` and `node-1:

**Primary broker config:**

```xml
<connectors>
  <!-- Connector used to be announced through cluster connections and notifications -->
  <connector name="node-0">tcp://localhost:61616</connector>
</connectors>
...<cluster-connections>
  <cluster-connection name="my-cluster">
    <connector-ref>node-0</connector-ref>
    ...<cluster-connection>
</cluster-connections>
```

**Backup broker config**

```xml
<connectors>
  <!-- Connector used to be announced through cluster connections and notifications -->
  <connector name="node-1">tcp://localhost:61617</connector>
</connectors>
<cluster-connections>
  <cluster-connection name="my-cluster">
    <connector-ref>node-1</connector-ref>
    ...<cluster-connection>
</cluster-connections>
```
36.5.3. Getting Notified of Connection Failure

JMS provides a standard mechanism for getting notified asynchronously of connection failure: java.jms.ExceptionListener. Please consult the JMS javadoc or any good JMS tutorial for more information on how to use this.

The Apache ActiveMQ Artemis core API also provides a similar feature in the form of the class org.apache.activemq.artemis.core.client.SessionFailureListener. Any ExceptionListener or SessionFailureListener instance will always be called by ActiveMQ Artemis on event of connection failure, irrespective of whether the connection was successfully failed over, reconnected or reattached, however you can find out if reconnect or reattach has happened by either the failedOver flag passed in on the connectionFailed on SessionFailureListener or by inspecting the error code on the javax.jms.JMSException which will be one of the following:

JMSException error codes:

**FAILOVER**

Failover has occurred and we have successfully reattached or reconnected.

**DISCONNECT**

No failover has occurred and we are disconnected.

36.5.4. Application-Level Failover

In some cases you may not want automatic client failover, and prefer to handle any connection failure yourself, and code your own manually reconnection logic in your own failure handler. We define this as application-level failover, since the failover is handled at the user application level.

To implement application-level failover, if you're using JMS then you need to set an ExceptionListener class on the JMS connection. The ExceptionListener will be called by Apache ActiveMQ Artemis in the event that connection failure is detected. In your ExceptionListener, you would close your old JMS connections, potentially look up new connection factory instances from JNDI and creating new connections.

For a working example of application-level failover, please see the Application-Layer Failover Example.

If you are using the core API, then the procedure is very similar: you would set a FailureListener on the core ClientSession instances.
Chapter 37. Network Isolation (Split Brain)

A split brain is a condition that occurs when two different brokers are serving the same messages at the same time. When this happens instead of client applications all sharing the same broker as they ought, they may become divided between the two split brain brokers. This is problematic because it can lead to:

- **Duplicate messages** e.g. when multiple consumers on the same JMS queue split between both brokers and receive the same message(s)
- **Missed messages** e.g. when multiple consumers on the same JMS topic split between both brokers and producers are only sending messages to one broker

Split brain most commonly happens when a pair of brokers in an HA replication configuration lose the replication connection linking them together. When this connection is lost the backup assumes that the primary has died and therefore activates. At this point there are two brokers on the network which are isolated from each other and since the backup has a copy of all the messages from the primary they are each serving the same messages.

What about shared store configurations?

While it is technically possible for split brain to happen with a pair of brokers in an HA shared store configuration it would require a failure in the file-locking mechanism of the storage device which the brokers are sharing.

One of the benefits of using a shared store is that the storage device itself acts as an arbiter to ensure consistency and mitigate split brain.

Recovering from a split brain may be as simple as stopping the broker which activated by mistake. However, this solution is only viable if no client application connected to it and performed messaging operations. The longer client applications are allowed to interact with split brain brokers the more difficult it will be to understand and remediate the resulting problems.

There are several different configurations you can choose from that will help mitigate split brain.

### 37.1. Pluggable Lock Manager

A pluggable lock manager configuration requires a 3rd party to establish a shared lock between primary and backup brokers. The shared lock ensures that either the primary or backup is active at any given point in time, similar to how the file lock functions in the shared storage use-case.

The plugin decides what 3rd party implementation is used. It could be something as simple as a shared file on a network file system that supports locking (e.g. NFS) or it could be something more complex like etcd.

The broker ships with a reference plugin implementation based on Apache ZooKeeper - a common implementation used for this kind of task.

The main benefit of a pluggable lock manager is that is releases the broker from the responsibility of establishing a reliable vote. This means that a single HA pair of brokers can be reliably protected.
37.2. Quorum Voting

Quorum voting is a process by which one node in a cluster can determine whether another node in the cluster is active without directly communicating with that node. Then the broker initiating the vote can take action based on the result (e.g. shutting itself down to avoid split-brain).

Quorum voting requires the participation of the other active brokers in the cluster. Of course this requires that there are, in fact, other active brokers in the cluster which means quorum voting won’t work with a single HA pair of brokers. Furthermore, it also won’t work with just two HA pairs of brokers either because that’s still not enough for a legitimate quorum. There must be at least three HA pairs to establish a proper quorum with quorum voting.

37.2.1. Voting Mechanics

When the replication connection between a primary and backup is lost the backup and/or the primary may initiate a vote.

For a vote to pass a majority of affirmative responses is required. For example, in a 3 node cluster a vote will pass with 2 affirmatives. For a 4 node cluster this would be 3 affirmatives and so on.

Backup Voting

By default, if a backup loses its replication connection to its primary it will activate automatically. However, it can be configured via the `vote-on-replication-failure` property to initiate a quorum vote in order to decide whether to activate or not. If this is done then the backup will keep voting until it either receives a vote allowing it to start or it detects that the primary is still active. In the latter case it will then restart as a backup.

See the section on Replication Configuration for more details on configuration.

Primary Voting

By default, if the primary server loses its replication connection to the backup then it will just carry on and wait for a backup to reconnect and start replicating again. However, this may mean that the primary remains active even though the backup has activated so this behavior is configurable via the `vote-on-replication-failure` property.

See the section on Replication Configuration for more details on configuration.

37.3. Pinging the network

You may configure one more addresses in `broker.xml` that that will be pinged throughout the life of the server. The server will stop itself if it can't ping one or more of the addresses in the list.

If you execute the `create` command using the `--ping` argument you will create a default XML that is
ready to be used with network checks:

```bash
$ ./artemis create /myDir/myServer --ping 10.0.0.1
```

This XML will be added to your `broker.xml`:

```xml
<network-check-period>10000</network-check-period>
<network-check-timeout>1000</network-check-timeout>
```

Once you lose connectivity towards 10.0.0.1 on the given example the broker will log something like this:

```
09:49:24,562 WARN [org.apache.activemq.artemis.core.server.NetworkHealthCheck] Ping Address /10.0.0.1 wasn't reacheable
09:49:36,577 INFO [org.apache.activemq.artemis.core.server.NetworkHealthCheck] Network is unhealthy, stopping service ActiveMQServerImpl::serverUUID=04fd5dd8-b18c-11e6-9efe-6a0001921ad0
09:49:36,625 INFO [org.apache.activemq.artemis.core.server] AMQ221002: Apache ActiveMQ Artemis Message Broker version 1.6.0 [04fd5dd8-b18c-11e6-9efe-6a0001921ad0] stopped, uptime 14.787 seconds
09:50:00,653 WARN [org.apache.activemq.artemis.core.server.NetworkHealthCheck] ping:
```
sendto: No route to host

09:50:10,656 WARN [org.apache.activemq.artemis.core.server.NetworkHealthCheck] Host is down: java.net.ConnectException: Host is down
  at java.net.Inet6AddressImpl.isReachable0(Native Method) [rt.jar:1.8.0_73]
  at java.net.Inet6AddressImpl.isReachable(Inet6AddressImpl.java:77) [rt.jar:1.8.0_73]
  at java.net.InetAddress.isReachable(InetAddress.java:502) [rt.jar:1.8.0_73]
  at org.apache.activemq.artemis.core.server.NetworkHealthCheck.run(NetworkHealthCheck.java:244) [artemis-commons-1.6.0-SNAPSHOT.jar:1.6.0-SNAPSHOT]
  at org.apache.activemq.artemis.core.server.ActiveMQScheduledComponent$2.run(ActiveMQScheduledComponent.java:189) [artemis-commons-1.6.0-SNAPSHOT.jar:1.6.0-SNAPSHOT]
  at org.apache.activemq.artemis.core.server.ActiveMQScheduledComponent$3.run(ActiveMQScheduledComponent.java:199) [artemis-commons-1.6.0-SNAPSHOT.jar:1.6.0-SNAPSHOT]
  at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511) [rt.jar:1.8.0_73]
  at java.util.concurrent.FutureTask.runAndReset(FutureTask.java:308) [rt.jar:1.8.0_73]
  at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$301(ScheduledThreadPoolExecutor.java:180) [rt.jar:1.8.0_73]
  at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:294) [rt.jar:1.8.0_73]
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142) [rt.jar:1.8.0_73]
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:617) [rt.jar:1.8.0_73]
  at java.lang.Thread.run(Thread.java:745) [rt.jar:1.8.0_73]

Once you reestablish your network connections towards the configured check-list:

Make sure you understand your network topology as this is meant to validate your network. Using IPs that could eventually disappear or be partially visible may defeat the purpose. You can use a list of multiple IPs. Any successful ping will make the server OK to continue running.
Chapter 38. Restart Sequence

Apache ActiveMQ Artemis ships with 2 architectures for providing HA features. The primary and backup brokers can be configured either using network replication or using shared storage. This document will share restart sequences for the brokers under various circumstances when the client applications are connected to it.

38.1. Restarting 1 broker at a time

When restarting the brokers one at a time at regular intervals, it is not important to follow any sequence. We just need to make sure that at least 1 broker in the primary/backup pair is active to take up the connections from the client applications.

While restarting the brokers while the client applications are connected kindly make sure that at least one broker is always active to serve the connected clients.

38.2. Completely shutting down the brokers and starting

If there is situation that we need to completely shutdown the brokers and start them again, please follow the following procedure:

1. Shut down all the backup brokers.
2. Shut down all the primary brokers.
3. Start all the primary brokers.
4. Start all the backup brokers.

This sequence is particularly important in case of network replication for the following reasons. If the primary broker is shutdown first the backup broker will activate and accept all the client connections. Then when the backup broker is stopped the clients will attempt to reconnect to the broker that was active most recently i.e. backup. Now, when we start the backup and primary brokers the clients will keep trying to connect to the last connection i.e. with backup and will never be able to connect until we restart the client applications. To avoid the hassle of restarting of client applications, we must follow the sequence as suggested above.

38.3. Split-brain situation

The following procedure helps the cluster to recover from the split-brain situation and getting the client connections auto-reconnected to the cluster. With this sequence, client applications do not need to be restarted in order to make connection with the brokers.

During the split brain situation both the primary and backup brokers are active and there is no replication that is happening from the primary broker to the backup.

In such situation, there can be some client applications that are connected to the primary broker
and other connected to the backup broker. Now after we restart the brokers and the cluster is properly formed.

Here, the clients that were connected to the primary broker during the split brain situation are auto-connected to the cluster and start processing the messages. But the clients that got connected to the backup broker are still trying to make connection with the broker. This happens because the backup broker has restarted in 'back up' mode.

Thus, not all the clients get connected to the brokers and function properly.

To avoid such mishap, kindly follow the below sequence:

1. Stop the backup broker
2. Start the backup broker. Observe the logs for the message "Waiting for the primary"
3. Stop the primary broker.
4. Start the primary broker. Observe the primary broker logs for "Server is active" Observe the backup broker logs for "backup announced"
5. Stop the primary broker again. Wait until the backup broker becomes live. Observe that all the clients are connected to the backup broker.
6. Start the primary broker. This time, all the connections will be switched to primary broker again,

During the split brain situation, messages are produced on the backup broker since it is live. While resolving the split brain situation, if there are some delta messages that are not produced on the backup broker. Those messages cannot be auto-recovered. There will be manual intervention required to retrieve the messages, sometime it is almost impossible to recover the messages. The above mentioned sequence helps in forming the cluster that was broken due to split brain and getting all the client applications to auto connected to the cluster without any need for client applications to be restarted.
Chapter 39. Activation Sequence Tools

You can use the Artemis CLI to execute activation sequence maintenance/recovery tools for Replication with Pluggable Lock Manager.

The 2 main commands are `activation list` and `activation set`, that can be used together to recover some disaster happened to local/coordinated activation sequences.

Here is a disaster scenario built around the RI (using Apache ZooKeeper and Apache curator) to demonstrate the usage of such commands.

39.1. ZooKeeper cluster disaster

A proper ZooKeeper cluster should use at least 3 nodes, but what happens if all these nodes crash loosing any activation state information required to manage replication?

During the disaster (i.e. ZooKeeper nodes are no longer reachable) the follow occurs:

- Active brokers shutdown (and if restarted, should hang waiting to reconnect to the ZooKeeper cluster again)
- Passive brokers unpair and wait to reconnect to the ZooKeeper cluster again

Necessary administrative action:

1. Stop all brokers
2. Restart ZooKeeper cluster
3. Search for brokers with the highest local activation sequence for their NodeID by running this command from the `bin` folder of the broker:

   ```
   $.artemis activation list --local
   Local activation sequence for NodeID=7debb3d1-0d4b-11ec-9704-ae9213b68ac4: 1
   ```

4. From the `bin` folder of the brokers with the highest local activation sequence

   ```
   # assuming 1 to be the highest local activation sequence obtained at the previous step
   # for NodeID 7debb3d1-0d4b-11ec-9704-ae9213b68ac4
   $.artemis activation set --remote --to 1
   Forced coordinated activation sequence for NodeID=7debb3d1-0d4b-11ec-9704-ae9213b68ac4 from 0 to 1
   ```

5. Restart all brokers: previously active ones should be able to be active again

The more ZooKeeper nodes there are the less chance that a disaster like this requires administrative intervention because it allows the ZooKeeper cluster to tolerate more failures.
Chapter 40. Connection Routers

Apache ActiveMQ Artemis connection routers allow incoming client connections to be distributed across multiple target brokers. The target brokers are grouped in pools and the connection routers use a key to select a target broker from a pool of brokers according to a policy.

40.1. Target Broker

Target broker is a broker that can accept incoming client connections and is local or remote. The local target is a special target that represents the same broker hosting the connection router. The remote target is another reachable broker.

40.2. Keys

The connection router uses a key to select a target broker. It is a string retrieved from an incoming client connection, the supported key types are:

- **CLIENT_ID**
  - is the JMS client ID.

- **SNI_HOST**
  - is the hostname indicated by the client in the SNI extension of the TLS protocol.

- **SOURCE_IP**
  - is the source IP address of the client.

- **USER_NAME**
  - is the username indicated by the client.

- **ROLE_NAME**
  - is a role associated with the authenticated user of the connection.

40.3. Pools

The pool is a group of target brokers with periodic checks on their state. It provides a list of ready target brokers to distribute incoming client connections only when it is active. A pool becomes active when the minimum number of target brokers, as defined by the quorum-size parameter, become ready. When it is not active, it doesn’t provide any target avoiding weird distribution at startup or after a restart. Including the local broker in the target pool allows broker hosting the router to accept incoming client connections as well. By default, a pool doesn’t include the local broker, to include it as a target the local-target-enabled parameter must be true. There are three pool types: cluster pool, discovery pool and static pool.

40.3.1. Cluster Pool

The cluster pool uses a cluster connection to get the target brokers to add. Let's take a look at a
cluster pool example from broker.xml that uses a cluster connection:

```xml
<pool>
  <cluster-connection>cluster1</cluster-connection>
</pool>
```

### 40.3.2. Discovery Pool

The discovery pool uses a **discovery group** to discover the target brokers to add. Let's take a look at a discovery pool example from broker.xml that uses a discovery group:

```xml
<pool>
  <discovery-group-ref discovery-group-name="dg1"/>
</pool>
```

### 40.3.3. Static Pool

The static pool uses a list of static connectors to define the target brokers to add. Let's take a look at a static pool example from broker.xml that uses a list of static connectors:

```xml
<pool>
  <static-connectors>
    <connector-ref>connector1</connector-ref>
    <connector-ref>connector2</connector-ref>
    <connector-ref>connector3</connector-ref>
  </static-connectors>
</pool>
```

### 40.3.4. Defining pools

A pool is defined by the **pool** element that includes the following items:

- the **username** element defines the username to connect to the target broker;
- the **password** element defines the password to connect to the target broker;
- the **check-period** element defines how often to check the target broker, measured in milliseconds, default is **5000**;
- the **quorum-size** element defines the minimum number of ready targets to activate the pool, default is **1**;
- the **quorum-timeout** element defines the timeout to get the minimum number of ready targets, measured in milliseconds, default is **3000**;
- the **local-target-enabled** element defines whether the pool has to include a local target, default is **false**;
- the **cluster-connection** element defines the **cluster connection** used by the **cluster pool**.
• the static-connectors element defines a list of static connectors used by the static pool;
• the discovery-group element defines the discovery group used by the discovery pool.

Let’s take a look at a pool example from broker.xml:

```xml
<pool>
  <quorum-size>2</quorum-size>
  <check-period>1000</check-period>
  <local-target-enabled>true</local-target-enabled>
  <static-connectors>
    <connector-ref>connector1</connector-ref>
    <connector-ref>connector2</connector-ref>
    <connector-ref>connector3</connector-ref>
  </static-connectors>
</pool>
```

### 40.4. Policies

The policy defines how to select a broker from a pool and allows key values transformation. The included policies are:

**FIRST_ELEMENT**

to select the first target broker from the pool which is ready. It is useful to select the ready target brokers according to the priority defined with their sequence order, ie supposing there are 2 target brokers this policy selects the second target broker only when the first target broker isn’t ready.

**ROUND_ROBIN**
to select a target sequentially from a pool, this policy is useful to evenly distribute;

**CONSISTENT_HASH**
to select a target by a key. This policy always selects the same target broker for the same key until it is removed from the pool.

**LEAST_CONNECTIONS**
to select the targets with the fewest active connections. This policy helps you maintain an equal distribution of active connections with the target brokers.

**CONSISTENT_HASH_MODULO** to transform a key value to a number from 0 to N-1, it takes a single `modulo` property to configure the bound N. One use case is CLIENT_ID sharding across a cluster of N brokers. With a consistent hash % N transformation, each client id can map exclusively to just one of the brokers.

A policy is defined by the policy element. Let’s take a look at a policy example from broker.xml:
40.5. Cache

The connection router provides a cache with a timeout to improve the stickiness of the target broker selected, returning the same target broker for a key value as long as it is present in the cache and is ready. So a connection router with the cache enabled doesn't strictly follow the configured policy. By default, the cache is not enabled.

A cache is defined by the `cache` element that includes the following items:

- the `persisted` element defines whether the cache has to persist entries, default is `false`;
- the `timeout` element defines the timeout before removing entries, measured in milliseconds, setting 0 will disable the timeout, default is `0`.

Let's take a look at a cache example from broker.xml:

```xml
<cache>
  <persisted>true</persisted>
  <timeout>60000</timeout>
</cache>
```

40.6. Defining connection routers

A connection router is defined by the `connection-router` element, it includes the following items:

- the `name` attribute defines the name of the connection router and is used to reference the router from an acceptor;
- the `key-type` element defines what type of key to select a target broker, the supported values are: `CLIENT_ID`, `SNI_HOST`, `SOURCE_IP`, `USER_NAME`, `ROLE_NAME`, default is `SOURCE_IP`, see `Keys` for further details;
- the `key-filter` element defines a regular expression to filter the resolved key values;
- the `local-target-filter` element defines a regular expression to match the key values that have to return a local target, the key value could be equal to the special string `NULL` if the value of the key is undefined or it doesn't match the `key-filter`;
- the `pool` element defines the pool to group the target brokers, see `pools`;
- the `policy` element defines the policy used to select the target brokers from the pool, see `policies`.

Let's take a look at some connection router examples from broker.xml:

```xml
<connection-routers>
  <connection-router name="local-partition">
```
40.7. Key values

The key value is retrieved from the incoming client connection. If the incoming client connection has no value for the key type used, the key value is set to the special string NULL. If the incoming client connection has a value for the key type used, the key value retrieved can be sequentially manipulated using a key-filter and a policy. If a key-filter is defined and the filter fails to match, the value is set to the special string NULL. If a policy with a key transformation is defined, the key value is set to the transformed value.

40.8. Connection Router Workflow

The connection router workflow include the following steps:
• Retrieve the key value from the incoming connection;
• Return the local target broker if the key value matches the local filter;
• Delegate to the pool:
  • Return the cached target broker if it is ready;
  • Get ready/active target brokers from the pool;
  • Select one target broker using the policy;
  • Add the selected broker in the cache;
• Return the selected broker.

Let's take a look at flowchart of the connection router workflow:
40.9. Data gravity

The first router configuration: local-partition, demonstrates the simplest use case, that of preserving data gravity by confining a subset of application data to a given broker. Each broker is given a subset of keys that it will exclusively service or reject. If brokers are behind a round-robin load-balancer or have full knowledge of the broker urls, their broker will eventually respond. The local-target-filter regular expression determines the granularity of partition that is best for preserving data gravity for your applications.

The challenge is in providing a consistent key in all related application connections.

the concept of data gravity tries to capture the reality that while addresses are shared by multiple applications, it is best to keep related addresses and their data co-located on a single broker. Typically, applications should connect to the data rather than the data moving to whatever broker the application connects too. This is particularly true when the amount of data (backlog) is large, the cost of movement to follow consumers outweighs the cost of delivery to the application. With the 'data gravity' mindset, operators are less concerned with numbers of connections and more concerned with applications and the addresses they need to interact with.

40.10. Redirection

Apache ActiveMQ Artemis provides a native redirection for supported clients and a new management API for other clients. The native redirection can be enabled per acceptor and is supported only for AMQP, CORE and OPENWIRE clients. The acceptor with the router url parameter will redirect the incoming connections. The router url parameter specifies the name of the connection router to use, ie the following acceptor will redirect the incoming CORE client connections using the connection router with the name simple-router:

```
<acceptor name="artemis">tcp://0.0.0.0:61616?router=simple-router;protocols=CORE</acceptor>
```

40.10.1. Native Redirect Sequence

The clients supporting the native redirection connect to the acceptor with the redirection enabled. The acceptor sends to the client the target broker to redirect if it is ready and closes the connection. The client connects to the target broker if it has received one before getting disconnected otherwise it connected again to the acceptor with the redirection enabled.
40.10.2. Management API Redirect Sequence

The clients not supporting the native redirection queries the management API of connection router to get the target broker to redirect. If the API returns a target broker the client connects to it otherwise the client queries again the API.
Chapter 41. Broker Connections

Instead of waiting for clients to connect, a broker can initiate a connection to another endpoint on a specific protocol.

Currently, this feature supports only the AMQP protocol. However, in the future, it might be expanded to other protocols.

Broker connections are configured by the `<broker-connections>` XML element in the `broker.xml` configuration file.

```xml
<broker-connections>
  ...
</broker-connections>
```

### 41.1. AMQP Server Connections

An ActiveMQ Artemis broker can initiate connections using the AMQP protocol. This means that the broker can connect to another AMQP server (not necessarily ActiveMQ Artemis) and create elements on that connection.

To define an AMQP broker connection, add an `<amqp-connection>` element within the `<broker-connections>` element in the `broker.xml` configuration file. For example:

```xml
<broker-connections>
  <amqp-connection uri="tcp://HOST:PORT" name="other-server" retry-interval="100"
                   reconnect-attempts="-1" user="john" password="doe">
    ...
  </amqp-connection>
</broker-connections>
```

**uri**

tcp://host:myport[?options] (this is a required argument)

**name**

Name of the connection used for management purposes

**user**

Username with which to connect to the endpoint (this is an optional argument)

**password**

Password with which to connect to the endpoint (this is an optional argument)

**retry-interval**

Time, in milliseconds to wait before retrying a connection after an error. The default value is 5000.
**reconnect-attempts**

default is -1 meaning infinite

**auto-start**

Should the broker connection start automatically with the broker. Default is true. If false it is necessary to call a management operation to start it.

The connection URI options for transport settings, such as enabling and configuring TLS, are common with other Artemis connector URIs. See the transport doc for more. An example configuration for a TLS AMQP broker-connection can be found in the broker examples at ./examples/features/broker-connection/amqp-sending-overssl.

If auto-start is disabled on the broker connection, the start of the broker connection will only happen after the management method startBrokerConnection(connectionName) is called on the ServerController.

The target endpoint needs permission for all operations that are configured. Therefore, if a security manager is being used, ensure that you perform the configured operations with a user with sufficient permissions.

### 41.2. AMQP Server Connection Operations

The following types of operations are supported on an AMQP server connection:

- **Mirrors**
  - The broker uses an AMQP connection to another broker and duplicates messages and sends acknowledgements over the wire.

- **Senders**
  - Messages received on specific queues are transferred to another endpoint.

- **Receivers**
  - The broker pulls messages from another endpoint.

- **Peers**
  - The broker creates both senders and receivers on another endpoint that knows how to handle them. This is currently implemented by Apache Qpid Dispatch.

### 41.3. Reconnecting and Failover

It is possible to determine how reconnection will happen on a broker connection.

These are the attributes that are available on the amqp-connection XML element:

**reconnect-attempts**

default is -1 (infinite). How many attempts will be done after a failed connection
retry-interval

default is 5000, in milliseconds, the wait between each retry of a connection

It is also possible to specify alternate hosts on a broker connection by appending a comma separated list after a # at the end of the URI. The broker connection would keep trying on the alternate list until one of the targets is available to connect. Example:

```
<broker-connections>
  <amqp-connection uri="tcp://ServerA:5672#BackupA:5672,BackupB:5672" name="ServerA"
    reconnect-attempts="-1" retry-interval="5000">
    ...
  </amqp-connection>
</broker-connections>
```

Figure 1. Broker Connection - Reconnecting and Failover.

The previous example portrays a case of connection failure towards ServerA. The system would try to connect to serverA, backupA, and backupB until it successfully connects to one of these nodes.

### 41.4. Mirroring

Mirroring will reproduce any operation that happened on the source brokers towards a target broker.

The following events are sent through mirroring:

- **Message sending**
  - Messages sent to one broker will be "replicated" to the target broker.

- **Message acknowledgement**
  - Acknowledgements removing messages at one broker will be sent to the target broker.
  - Note that if the message is pending for a consumer on the target mirror, the ack will not succeed and the message might be delivered by both brokers.

- **Queue and address creation.**
- **Queue and address deletion.**

By default, every operation is sent asynchronously without blocking any clients. However, if you set `sync=true` on the mirror configuration the clients will always wait for the mirror on every blocking operation.
41.4.1. Mirror configuration

Add a `<mirror>` element within the `<amqp-connection>` element to configure mirroring to the target broker.

The following optional arguments can be utilized:

**message-acknowledgements**

Specifies whether message acknowledgements are sent. The default value is true.

**queue-creation**

Specifies whether a queue- or address-creation event is sent. The default value is true.

**queue-removal**

Specifies whether a queue- or address-removal event is sent. The default value is true.

**address-filter**

An optional comma-separated list of inclusion and/or exclusion filter entries used to govern which addresses (and related queues) mirroring events will be created for on this broker-connection. That is, events will only be mirrored to the target broker for addresses that match the filter. An address is matched when it begins with an inclusion entry specified in this field, unless the address is also explicitly excluded by another entry.

An exclusion entry is prefixed with `!` to denote any address beginning with that value does not match. If no inclusion entry is specified in the list, all addresses not explicitly excluded will match. If the address-filter attribute is not specified, then all addresses (and related queues) will match and be mirrored.

Examples:

- eu matches all addresses starting with eu
- !eu matches all address except for those starting with eu
- eu.uk, eu.de matches all addresses starting with either eu.uk or eu.de
- eu,!eu.uk matches all addresses starting with eu but not those starting with eu.uk

- Address exclusion will always take precedence over address inclusion.
- Address matching on mirror elements is prefix-based and does not support wild-card matching.

**sync**

If set to true any client blocking operation will be held until the mirror has confirmed receiving the operation. Default is false.

- Notice that a disconnected node would hold all operations from the client. If you set `sync=true` you must reconnect a mirror before performing any operations.

An example of a mirror configuration is shown below:
41.4.2. Store and Forward Queue

Mirror events are always stored on a local queue prefixed as $ACTIVEMQ_ARTEMIS_MIRROR_ and then concatenated with the broker connection's configured name.

So, in the following configuration, mirror events will be stored on a queue named $ACTIVEMQ_ARTEMIS_MIRROR_brokerB.

These messages are then transferred to brokerB:5672. A producer to the address $ACTIVEMQ_ARTEMIS_MIRROR_brokerB will be created towards brokerB. If there is a security manager configured, security roles must be provided to the user on the broker connection.

Notice the queue $ACTIVEMQ_ARTEMIS_MIRROR_brokerB will not actually exist on brokerB and so it will not be visible on the administration console. The target broker will treat these messages accordingly as mirror events and perform the appropriate operations at the target broker.

41.4.3. Pre Existing Messages

The broker will only mirror messages arriving from the point in time the mirror was configured. Previously existing messages will not be forwarded to other brokers.

41.5. Dual Mirror (Disaster Recovery)

ActiveMQ Artemis supports automatic fallback mirroring. Every sent message and every acknowledgement is asynchronously replicated to the mirrored broker.

On the following diagram there will be two servers called DataCenter1, and DataCenter2. In order to have a dual mirror configuration it is necessary to add the mirror broker connection on each broker.xml:
On *DataCenter1* the following code should be added on *broker.xml*:

```xml
<broker-connections>
  <amqp-connection uri="tcp://DataCenter2:5672" name="DC2">
    <mirror/>
  </amqp-connection>
</broker-connections>
```

The following xml snippet should be added on *DataCenter2*'s *broker.xml*:

```xml
<broker-connections>
  <amqp-connection uri="tcp://DataCenter1:5672" name="DC1">
    <mirror/>
  </amqp-connection>
</broker-connections>
```

The broker connections will replicate sends and acknowledgements to the other broker, no matter where they originated. If messages are sent on DC1 (*DataCenter1*) these will be automatically transferred to DC2 (*DataCenter2*). Message acknowledgements received on DC2 will be automatically related back to DC1.

The only exception to that rule would be if there were already consumers with pending messages on any server, where a mirrored acknowledgement will not prevent the message being consumed by both consumers.

> It is recommended to not have active consumers on both servers.

### 41.5.1. Example

There is an example as part of the distribution showing dual broker configuration (or disaster recovery) in `./examples/features/broker-connection/disaster-recovery`.

On the provided example above, two brokers are configured to mirror each other and whatever happens in one broker is immediately copied over to the other broker.

### 41.6. Senders and Receivers

It is possible to connect an ActiveMQ Artemis broker to another AMQP endpoint simply by creating a sender or receiver broker connection element.
For a sender, the broker creates a message consumer on a queue that sends messages to another AMQP endpoint.

For a receiver, the broker creates a message producer on an address that receives messages from another AMQP endpoint.

Both elements function as a message bridge. However, there is no additional overhead required to process messages. Senders and receivers behave just like any other consumer or producer in ActiveMQ Artemis.

Specific queues can be configured by senders or receivers. Wildcard expressions can be used to match senders and receivers to specific addresses or sets of addresses. When configuring a sender or receiver, the following properties can be set:

- **address-match**
  Match the sender or receiver to a specific address or set of addresses, using a wildcard expression.

- **queue-name**
  Configure the sender or receiver for a specific queue.

### 41.6.1. Examples

Using address expressions:

```xml
<broker-connections>
  <amqp-connection uri="tcp://HOST:PORT" name="other-server">
    <sender address-match="queues.#"/>
    <!-- notice the local queues for remotequeues.# need to be created on this broker -->
    <receiver address-match="remotequeues.#"/>
  </amqp-connection>
</broker-connections>

<addresses>
  <address name="remotequeues.A">
    <anycast>
      <queue name="remoteQueueA"/>
    </anycast>
  </address>
  <address name="queues.B">
    <anycast>
      <queue name="localQueueB"/>
    </anycast>
  </address>
</addresses>
```

Using queue names:
Receivers can only be matched to a local queue that already exists. Therefore, if receivers are being used, ensure that queues are pre-created locally. Otherwise, the broker cannot match the remote queues and addresses.

Do not create a sender and a receiver to the same destination. This creates an infinite loop of sends and receives.

41.7. Peers

The broker can be configured as a peer which connects to the Apache Qpid Dispatch Router and instructs it that the broker will act as a store-and-forward queue for a given AMQP waypoint address configured on the router. In this scenario, clients connect to a router to send and receive messages using a waypointed address, and the router routes these messages to or from the queue on the broker.

The peer configuration causes ActiveMQ Artemis to create a sender and receiver pair for each destination matched in the broker-connection configuration, with these carrying special configuration to let Qpid Dispatch know to collaborate with the broker. This replaces the traditional need of a router-initiated connection and auto-links.

Qpid Dispatch Router offers a lot of advanced networking options that be used together with ActiveMQ Artemis.

With a peer configuration, the same properties are present as when there are senders and receivers. For example, a configuration where queues with names beginning queue. act as storage for the matching router waypoint address would be:
<broker-connections>
  <amqp-connection uri="tcp://HOST:PORT" name="router">
    <peer address-match="queues.#"/>
  </amqp-connection>
</broker-connections>

<addresses>
  <address name="queues.A">
    <anycast>
      <queue name="queues.A"/>
    </anycast>
  </address>
  <address name="queues.B">
    <anycast>
      <queue name="queues.B"/>
    </anycast>
  </address>
</addresses>

There must be a matching address waypoint configuration on the router instructing it that the particular router addresses the broker attaches to should be treated as waypoints. For example, a similar prefix-based router address configuration would be:

```
address {
  prefix: queue
  waypoint: yes
}
```

For more information refer to the "brokered messaging" documentation for Apache Qpid Dispatch Router.

⚠️ Do not use this feature to connect to another broker, otherwise any message sent will be immediately ready to consume creating an infinite echo of sends and receives.

⚠️ It is not necessary to configure the router with a connector or auto-links to communicate with the broker. The brokers peer configuration replaces these aspects of the router waypoint usage.

### 41.8. Address Consideration

It is highly recommended that `address name` and `queue name` are the same. When a queue with its distinct name (as in the following example) is used senders and receivers will always use the `address name` when creating the remote endpoint.
The above example would create an AMQP sender towards queues.A.

To avoid confusion it is recommended that address name and queue name are kept the same.

41.9. Federation

Broker federation allows the local broker to create remote receivers for addresses or queues that have local demand. Conversely, the broker connection can send federation configuration to the remote broker causing it to create receivers on the local broker based on remote demand on an address or queue over this same connection.

Add a <federation> element within the <amqp-connection> element to configure federation to the broker instance. The <amqp-connection> contains all the configuration for authentication and reconnection handling. See above sections to configure those values.

The broker connection federation configuration consists of one or more policies that define either local or remote federation configurations for addresses or queues.
41.9.1. Local and remote address federation

Local or Remote address federation configures the local or remote broker to watch for demand on addresses. When demand exists it will create a consumer on the matching address on the opposing broker. Because the consumers are created on addresses on the opposing broker, the authentication credentials supplied to the broker connection must have sufficient access to read from (local address federation) or write to the matching address (remote address federation) on the opposing broker.

An example of address federation configuration is shown below. Local and remote address policies have identical configuration parameters, only the policy element names are different.

```xml
<broker-connections>
  <amqp-connection uri="tcp://HOST:PORT" name="federation-example">
    <federation>
      <local-address-policy name="example-local-address-policy"
        auto-delete="false"
        auto-delete-delay="1"
        auto-delete-message-count="12"
        max-hops="1"
        enable-divert-bindings="true">
        <include address-match="local-address.#" />
        <exclude address-match="local-address.excluded" />
      </local-address-policy>
    </federation>
  </amqp-connection>
</broker-connections>
```

**name**

The name of the policy, these names should be unique within a broker connection’s federation policy elements.

**include**

The address-match pattern to use to match included addresses, multiple of these can be set. If none are set all addresses are matched.

**exclude**

The address-match pattern to use to match excluded addresses, multiple of these can be set, or it can be omitted if no excludes are needed.

**max-hops**

The maximum number of hops that a message can perform to be federated. Default value is 0 and will work for most simple federation deployments. However, in certain topologies a higher value may be required to prevent looping.
auto-delete

For address federation, a durable queue is created on the broker from which messages are being federated. Set this property to true to mark the queue for automatic deletion once the initiating broker disconnects and the delay and message count parameters are met. This is a useful option if you want to automate the cleanup of dynamically-created queues. The default value is false, which means that the queue is not automatically deleted.

auto-delete-delay

The amount of time in milliseconds after the initiating broker has disconnected before the created queue can be eligible for auto-delete. The default value is 0 when option is omitted.

auto-delete-message-count

After the initiating broker has disconnected, the maximum number of messages allowed in the remote queue for the queue to be eligible for automatic deletion. The default value is 0.

enable-divert-bindings

Setting to true enables divert bindings to be listened-to for demand. If a divert binding with an address matches the included addresses for the address policy, any queue bindings that match the forwarding address of the divert creates demand. The default value is false.

41.9.2. Local and remote queue federation

Local or remote queue federation configures the local or remote broker to watch for demand on queues and when demand exists it will create a consumer on the matching queue on the opposing broker. Because the consumers are created on queues on the opposing broker, the authentication credentials supplied to the broker connection must have sufficient access to read from (local queue federation) or write to the matching queue (remote queue federation) on the opposing broker.

An example of queue federation configuration is shown below. Local and remote queue policies have identical configuration parameters, only the policy element names are different.

```xml
<broker-connections>
  <amqp-connection uri="tcp://HOST:PORT" name="federation-example">
    <federation>
      <local-queue-policy name="example-local-queue-policy">
        <include address-match="#" queue-match="remote-queue"/>
        <exclude address-match="excluded.#" queue-match="remote-queue-excluded"/>
      </local-queue-policy>
    </federation>
  </amqp-connection>
</broker-connections>
```

name

The name of the policy. These names should be unique within a broker connection’s federation policy elements.
include
The queue-match pattern to use to match included queues. Multiple of these can be set. If none are set all queues are matched.

exclude
The queue-match pattern to use to match excluded queues. Multiple of these can be set, or it can be omitted if no excludes are needed.

priority-adjustment
When federation consumers are created this value can be used to ensure that those federation consumers have a lower priority value than other local consumers on the same queue. The default is -1.

include-federated
Controls if consumers on a queue which come from federation instances should be counted when observing a queue for demand, by default this value is false and federation consumers are not counted.
Chapter 42. The JMS Bridge

Apache ActiveMQ Artemis includes a fully functional JMS message bridge.

The function of the bridge is to consume messages from a source queue or topic, and send them to a target queue or topic, typically on a different server.

The JMS Bridge is not intended as a replacement for transformation and more expert systems such as Camel. The JMS Bridge may be useful for fast transfers as this chapter covers, but keep in mind that more complex scenarios requiring transformations will require you to use a more advanced transformation system that will play on use cases that will go beyond Apache ActiveMQ Artemis.

The source and target servers do not have to be in the same cluster which makes bridging suitable for reliably sending messages from one cluster to another, for instance across a WAN, and where the connection may be unreliable.

A bridge can be deployed as a standalone application or as a web application managed by the embedded Jetty instance bootstrapped with Apache ActiveMQ Artemis. The source and the target can be located in the same virtual machine or another one.

The bridge can also be used to bridge messages from other non Apache ActiveMQ Artemis JMS servers, as long as they are JMS 1.1 compliant.

Do not confuse a JMS bridge with a core bridge. A JMS bridge can be used to bridge any two JMS 1.1 compliant JMS providers and uses the JMS API. A core bridge is used to bridge any two Apache ActiveMQ Artemis instances and uses the core API. Always use a core bridge if you can in preference to a JMS bridge. The core bridge will typically provide better performance than a JMS bridge. Also the core bridge can provide once and only once delivery guarantees without using XA.

The bridge has built-in resilience to failure so if the source or target server connection is lost, e.g. due to network failure, the bridge will retry connecting to the source and/or target until they come back online. When it comes back online it will resume operation as normal.

The bridge can be configured with an optional JMS selector, so it will only consume messages matching that JMS selector

It can be configured to consume from a queue or a topic. When it consumes from a topic it can be configured to consume using a non durable or durable subscription

The JMS Bridge is a simple POJO so can be deployed with most frameworks, simply instantiate the org.apache.activemq.artemis.api.jms.bridge.impl.JMSBridgeImpl class and set the appropriate parameters.

42.1. JMS Bridge Parameters

The main POJO is the JMSBridge. It is configurable by the parameters passed to its constructor.
• Source Connection Factory Factory

This injects the SourceCFF bean (also defined in the beans file). This bean is used to create the source ConnectionFactory

• Target Connection Factory Factory

This injects the TargetCFF bean (also defined in the beans file). This bean is used to create the target ConnectionFactory

• Source Destination Factory Factory

This injects the SourceDestinationFactory bean (also defined in the beans file). This bean is used to create the source Destination

• Target Destination Factory Factory

This injects the TargetDestinationFactory bean (also defined in the beans file). This bean is used to create the target Destination

• Source User Name

  this parameter is the username for creating the source connection

• Source Password

  this parameter is the parameter for creating the source connection

• Target User Name

  this parameter is the username for creating the target connection

• Target Password

  this parameter is the password for creating the target connection

• Selector

  This represents a JMS selector expression used for consuming messages from the source destination. Only messages that match the selector expression will be bridged from the source to the target destination

  The selector expression must follow the JMS selector syntax

• Failure Retry Interval

  This represents the amount of time in ms to wait between trying to recreate connections to the source or target servers when the bridge has detected they have failed

• Max Retries

  This represents the number of times to attempt to recreate connections to the source or target servers when the bridge has detected they have failed. The bridge will give up after trying this
• Quality Of Service

This parameter represents the desired quality of service mode

Possible values are:

  ◦ AT_MOST_ONCE
  ◦ DUPLICATES_OK
  ◦ ONCE_AND_ONLY_ONCE

See Quality Of Service section for an explanation of these modes.

• Max Batch Size

This represents the maximum number of messages to consume from the source destination before sending them in a batch to the target destination. Its value must == 1

• Max Batch Time

This represents the maximum number of milliseconds to wait before sending a batch to target, even if the number of messages consumed has not reached MaxBatchSize. Its value must be -1 to represent 'wait forever', or >= 1 to specify an actual time

• Subscription Name

If the source destination represents a topic, and you want to consume from the topic using a durable subscription then this parameter represents the durable subscription name

• Client ID

If the source destination represents a topic, and you want to consume from the topic using a durable subscription then this attribute represents the JMS client ID to use when creating/looking up the durable subscription

• Add MessageID In Header

If true, then the original message's message ID will be appended in the message sent to the destination in the header ACTIVEMQ_BRIDGE_MSG_ID_LIST. If the message is bridged more than once, each message ID will be appended. This enables a distributed request-response pattern to be used when you receive the message you can send back a response using the correlation id of the first message id, so when the original sender gets it back it will be able to correlate it.

• MBean Server

To manage the JMS Bridge using JMX, set the MBeanServer where the JMS Bridge MBean must be registered (e.g. the JVM Platform MBeanServer)
• ObjectName

If you set the MBeanServer, you also need to set the ObjectName used to register the JMS Bridge MBean (must be unique)

The "transactionManager" property points to a JTA transaction manager implementation and should be set if you need to use the 'ONCE_AND_ONCE_ONLY' Quality of Service. Apache ActiveMQ Artemis doesn't ship with such an implementation, but if you are running within an Application Server you can inject the Transaction Manager that is shipped.

### 42.2. Source and Target Connection Factories

The source and target connection factory factories are used to create the connection factory used to create the connection for the source or target server.

The configuration example above uses the default implementation provided by Apache ActiveMQ Artemis that looks up the connection factory using JNDI. For other Application Servers or JMS providers a new implementation may have to be provided. This can easily be done by implementing the interface `org.apache.activemq.artemis.jms.bridge.ConnectionFactoryFactory`.

### 42.3. Source and Target Destination Factories

Again, similarly, these are used to create or lookup up the destinations.

In the configuration example above, we have used the default provided by Apache ActiveMQ Artemis that looks up the destination using JNDI.

A new implementation can be provided by implementing `org.apache.activemq.artemis.jms.bridge.DestinationFactory` interface.

### 42.4. Quality Of Service

The quality of service modes used by the bridge are described here in more detail.

#### 42.4.1. AT_MOST_ONCE

With this QoS mode messages will reach the destination from the source at most once. The messages are consumed from the source and acknowledged before sending to the destination. Therefore there is a possibility that if failure occurs between removing them from the source and them arriving at the destination they could be lost. Hence delivery will occur at most once.

This mode is available for both durable and non-durable messages.

#### 42.4.2. DUPLICATES_OK

With this QoS mode, the messages are consumed from the source and then acknowledged after they have been successfully sent to the destination. Therefore there is a possibility that if failure occurs after sending to the destination but before acknowledging them, they could be sent again when the
system recovers. i.e. the destination might receive duplicates after a failure.

This mode is available for both durable and non-durable messages.

### 42.4.3. **ONCE_AND_ONLY_ONCE**

This QoS mode ensures messages will reach the destination from the source once and only once. (Sometimes this mode is known as "exactly once"). If both the source and the destination are on the same Apache ActiveMQ Artemis server instance then this can be achieved by sending and acknowledging the messages in the same local transaction. If the source and destination are on different servers this is achieved by enlisting the sending and consuming sessions in a JTA transaction. The JTA transaction is controlled by a JTA Transaction Manager which will need to be set via the settransactionManager method on the Bridge.

This mode is only available for durable messages.

For a specific application it may possible to provide once and only once semantics without using the ONCE_AND_ONLY_ONCE QoS level. This can be done by using the DUPLICATES_OK mode and then checking for duplicates at the destination and discarding them. Some JMS servers provide automatic duplicate message detection functionality, or this may be possible to implement on the application level by maintaining a cache of received message ids on disk and comparing received messages to them. The cache would only be valid for a certain period of time so this approach is not as watertight as using ONCE_AND_ONLY_ONCE but may be a good choice depending on your specific application.

### 42.4.4. **Time outs and the JMS bridge**

There is a possibility that the target or source server will not be available at some point in time. If this occurs then the bridge will try Max Retries to reconnect every Failure Retry Interval milliseconds as specified in the JMS Bridge definition.

If you implement your own factories for looking up JMS resources then you will have to bear in mind timeout issues.

### 42.4.5. **Examples**

Please see [JMS Bridge Example](#) which shows how to programmatically instantiate and configure a JMS Bridge to send messages to the source destination and consume them from the target destination between two standalone Apache ActiveMQ Artemis brokers.
Chapter 43. Authentication & Authorization

This chapter describes how security works with Apache ActiveMQ Artemis and how you can configure it.

To disable security completely simply set the `security-enabled` property to `false` in the `broker.xml` file.

For performance reasons both authentication and authorization is cached independently. Entries are removed from the caches (i.e. invalidated) either when the cache reaches its maximum size in which case the least-recently used entry is removed or when an entry has been in the cache "too long".

The size of the caches are controlled by the `authentication-cache-size` and `authorization-cache-size` configuration parameters. Both default to 1000.

How long cache entries are valid is controlled by `security-invalidation-interval`, which is in milliseconds. Using 0 will disable caching. The default is 10000 ms.

43.1. Tracking the Validated User

To assist in security auditing the `populate-validated-user` option exists. If this is `true` then the server will add the name of the validated user to the message using the key `_AMQ_VALIDATED_USER`. For JMS and Stomp clients this is mapped to the key `JMSXUserID`. For users authenticated based on their SSL certificate this name is the name to which their certificate's DN maps. If `security-enabled` is `false` and `populate-validated-user` is `true` then the server will simply use whatever user name (if any) the client provides. This option is `false` by default.

43.2. Role based security for addresses

Apache ActiveMQ Artemis contains a flexible role-based security model for applying security to queues, based on their addresses.

As explained in Using Core, Apache ActiveMQ Artemis core consists mainly of sets of queues bound to addresses. A message is sent to an address and the server looks up the set of queues that are bound to that address, the server then routes the message to those set of queues.

Apache ActiveMQ Artemis allows sets of permissions to be defined against the queues based on their address. An exact match on the address can be used or a wildcard match can be used.

There are different permissions that can be given to the set of queues which match the address. Those permissions are:

**createAddress**

This permission allows the user to create an address fitting the `match`.

**deleteAddress**

This permission allows the user to delete an address fitting the `match`. 
createDurableQueue
This permission allows the user to create a durable queue under matching addresses.

deleteDurableQueue
This permission allows the user to delete a durable queue under matching addresses.

createNonDurableQueue
This permission allows the user to create a non-durable queue under matching addresses.

deleteNonDurableQueue
This permission allows the user to delete a non-durable queue under matching addresses.

send
This permission allows the user to send a message to matching addresses.

consume
This permission allows the user to consume a message from a queue bound to matching addresses.

browse
This permission allows the user to browse a queue bound to the matching address.

manage
This permission allows the user to invoke management operations by sending management messages to the management address.

The following two permissions pertain to operations on the management apis of the broker. They split management operations into two sets, read only for view, and edit for mutating operations. The split is controlled by a regular expression. Methods that match will require the view permission, all others require edit. The regular expression can be modified through the configuration attribute view-permission-method-match-pattern. These permissions are applicable to the management address and to MBean access. They are granted to match addresses prefixed with the management prefix.

view
This permission allows access to a read-only subset of management operations.

edit
This permission allows access to the mutating management operations, any operation not in the view set.

For each permission, a list of roles who are granted that permission is specified. If the user has any of those roles, he/she will be granted that permission for that set of addresses.

Let's take a simple example, here's a security block from broker.xml file:

```
<security-setting match="globalqueues.europe.#">
  <permission type="createDurableQueue" roles="admin"/>
</security-setting>
```
Using the default wildcard syntax the # character signifies "any sequence of words". Words are delimited by the . character. Therefore, the above security block applies to any address that starts with the string "globalqueues.europe."

Only users who have the admin role can create or delete durable queues bound to an address that starts with the string "globalqueues.europe."

Any users with the roles admin, guest, or europe-users can create or delete temporary queues bound to an address that starts with the string "globalqueues.europe."

Any users with the roles admin or europe-users can send messages to these addresses or consume messages from queues bound to an address that starts with the string "globalqueues.europe."

The mapping between a user and what roles they have is handled by the security manager. Apache ActiveMQ Artemis ships with a user manager that reads user credentials from a file on disk, and can also plug into JAAS or JBoss Application Server security.

For more information on configuring the security manager, please see 'Changing the Security Manager'.

There can be zero or more security-setting elements in each xml file. Where more than one match applies to a set of addresses the more specific match takes precedence.

Let's look at an example of that, here's another security-setting block:

```xml
<security-setting match="globalqueues.europe.orders.#">
  <permission type="send" roles="europe-users"/>
  <permission type="consume" roles="europe-users"/>
</security-setting>
```

In this security-setting block the match globalqueues.europe.orders.# is more specific than the previous match globalqueues.europe.#. So any addresses which match globalqueues.europe.orders.# will take their security settings only from the latter security-setting block.

Note that settings are not inherited from the former block. All the settings will be taken from the more specific matching block, so for the address globalqueues.europe.orders.plastics the only permissions that exist are send and consume for the role europe-users. The permissions createDurableQueue, deleteDurableQueue, createNonDurableQueue, deleteNonDurableQueue are not inherited from the other security-setting block.

By not inheriting permissions, it allows you to effectively deny permissions in more specific security-setting blocks by simply not specifying them. Otherwise it would not be possible to deny
permissions in sub-groups of addresses.

43.2.1. Fine-grained security using fully qualified queue name

In certain situations it may be necessary to configure security that is more fine-grained that simply across an entire address. For example, consider an address with multiple queues:

```
<addresses>
    <address name="foo">
        <anycast>
            <queue name="q1" />
            <queue name="q2" />
        </anycast>
    </address>
</addresses>
```

You may want to limit consumption from q1 to one role and consumption from q2 to another role. You can do this using the fully qualified queue name (i.e. FQQN) in the match of the security-setting, e.g.:

```
<security-setting match="foo::q1">
    <permission type="consume" roles="q1Role"/>
</security-setting>
<security-setting match="foo::q2">
    <permission type="consume" roles="q2Role"/>
</security-setting>
```

Wildcard matching doesn’t work in conjunction with FQQN. The explicit goal of using FQQN here is to be exact.

43.2.2. Applying view and edit permissions to the management api

The view and edit permissions are optionally applied to the management apis of the broker.

For RBAC on JMX MBean access they can replace the authorization section in management.xml as described at JMX authorization in broker.xml

For RBAC on management resources accessed via messages sent to the management address, the additional permissions are enabled by configuring management-message-rbac as described at Fine grained RBAC on management messages

The split between operations that require the view and edit permissions can be controlled via view-permission-method-match-pattern

43.3. Security Setting Plugin

Aside from configuring sets of permissions via XML these permissions can alternatively be
configured via a plugin which implements org.apache.activemq.artemis.core.server.SecuritySettingPlugin e.g:

```xml
<security-settings>
  <security-setting-plugin class-name="org.apache.activemq.artemis.core.server.impl.LegacyLDAPSecuritySettingPlugin">
    <setting name="initialContextFactory" value="com.sun.jndi.ldap.LdapCtxFactory"/>
    <setting name="connectionURL" value="ldap://localhost:1024"/>
    <setting name="connectionUsername" value="uid=admin,ou=system"/>
    <setting name="connectionPassword" value="secret"/>
    <setting name="connectionProtocol" value="s"/>
    <setting name="authentication" value="simple"/>
  </security-setting-plugin>
</security-settings>
```

Most of this configuration is specific to the plugin implementation. However, there are two configuration details that will be specified for every implementation:

**class-name**

This attribute of `security-setting-plugin` indicates the name of the class which implements org.apache.activemq.artemis.core.server.SecuritySettingPlugin.

**setting**

Each of these elements represents a name/value pair that will be passed to the implementation for configuration purposes.

See the JavaDoc on org.apache.activemq.artemis.core.server.SecuritySettingPlugin for further details about the interface and what each method is expected to do.

### 43.3.1. Available plugins

**LegacyLDAPSecuritySettingPlugin**

This plugin will read the security information that was previously handled by LDAPAuthorizationMap and the cachedLDAPAuthorizationMap in Apache ActiveMQ "Classic" and turn it into Artemis security settings where possible. The security implementations of ActiveMQ "Classic" and Artemis don't match perfectly so some translation must occur to achieve near equivalent functionality.

Here is an example of the plugin's configuration:

```xml
<security-setting-plugin class-name="org.apache.activemq.artemis.core.server.impl.LegacyLDAPSecuritySettingPlugin">
  <setting name="initialContextFactory" value="com.sun.jndi.ldap.LdapCtxFactory"/>
  <setting name="connectionURL" value="ldap://localhost:1024"/>
  <setting name="connectionUsername" value="uid=admin,ou=system"/>
  <setting name="connectionPassword" value="secret"/>
  <setting name="connectionProtocol" value="s"/>
  <setting name="authentication" value="simple"/>
</security-setting-plugin>
```
The implementation is org.apache.activemq.artemis.core.server.impl.LegacyLDAPSecuritySettingPlugin.

**initialContextFactory**

The initial context factory used to connect to LDAP. It must always be set to com.sun.jndi.ldap.LdapCtxFactory (i.e. the default value).

**connectionURL**

Specifies the location of the directory server using an ldap URL, ldap://Host:Port. You can optionally qualify this URL, by adding a forward slash, /, followed by the DN of a particular node in the directory tree. For example, ldap://ldapserver:10389/ou=system. The default is ldap://localhost:1024.

**connectionUsername**

The DN of the user that opens the connection to the directory server. For example, uid=admin,ou=system. Directory servers generally require clients to present username/password credentials in order to open a connection.

**connectionPassword**

The password that matches the DN from connectionUsername. In the directory server, in the DIT, the password is normally stored as a userPassword attribute in the corresponding directory entry.

**connectionProtocol**

Currently the only supported value is a blank string. In future, this option will allow you to select the Secure Socket Layer (SSL) for the connection to the directory server.

This option must be set explicitly to an empty string, because it has no default value.

**authentication**

Specifies the authentication method used when binding to the LDAP server. Can take either of the values, simple (username and password, the default value) or none (anonymous).

Simple Authentication and Security Layer (SASL) authentication is currently not supported.

**destinationBase**

Specifies the DN of the node whose children provide the permissions for all destinations. In this case the DN is a literal value (that is, no string substitution is performed on the property value). For example, a typical value of this property is ou=destinations,o=ActiveMQ,ou=system (i.e. the default value).

**filter**

Specifies an LDAP search filter, which is used when looking up the permissions for any kind of
destination. The search filter attempts to match one of the children or descendants of the queue or topic node. The default value is (cn=*).

**roleAttribute**

Specifies an attribute of the node matched by filter, whose value is the DN of a role. Default value is uniqueMember.

**adminPermissionValue**

Specifies a value that matches the admin permission. The default value is admin.

**readPermissionValue**

Specifies a value that matches the read permission. The default value is read.

**writePermissionValue**

Specifies a value that matches the write permission. The default value is write.

**enableListener**

Whether or not to enable a listener that will automatically receive updates made in the LDAP server and update the broker's authorization configuration in real-time. The default value is true.

Some LDAP servers (e.g. OpenLDAP) don't support the "persistent search" feature which allows the "listener" functionality to work. For these servers set the refreshInterval to a value greater than 0.

**refreshInterval**

How long to wait (in seconds) before refreshing the security settings from the LDAP server. This can be used for LDAP servers which don't support the "persistent search" feature needed for use with enableListener (e.g. OpenLDAP). Default is 0 (i.e. no refresh).

Keep in mind that this can be a potentially expensive operation based on how often the refresh is configured and how large the data set is so take care in how refreshInterval is configured.

**mapAdminToManage**

Whether or not to map the legacy admin permission to the manage permission. See details of the mapping semantics below. The default value is false.

**allowQueueAdminOnRead**

Whether or not to map the legacy read permission to the createDurableQueue, createNonDurableQueue, and deleteDurableQueue permissions so that JMS clients can create durable and non-durable subscriptions without needing the admin permission. This was allowed in ActiveMQ "Classic". The default value is false.

The name of the queue or topic defined in LDAP will serve as the "match" for the security-setting, the permission value will be mapped from the ActiveMQ "Classic" type to the Artemis type, and the role will be mapped as-is.

ActiveMQ "Classic" only has 3 permission types - read, write, and admin. These permission types are described on their website. However, as described previously, ActiveMQ Artemis has 9 permission
types - `createAddress`, `deleteAddress`, `createDurableQueue`, `deleteDurableQueue`, `createNonDurableQueue`, `deleteNonDurableQueue`, `send`, `consume`, `browse`, and `manage`. Here's how the old types are mapped to the new types:

**read**
- `consume`, `browse`

**write**
- `send`

**admin**
- `createAddress`, `deleteAddress`, `createDurableQueue`, `deleteDurableQueue`, `createNonDurableQueue`, `deleteNonDurableQueue`, `manage` (if `mapAdminToManage` is true)

As mentioned, there are a few places where a translation was performed to achieve some equivalence:

- This mapping doesn't include the Artemis `manage` permission type by default since there is no type analogous for that in ActiveMQ "Classic". However, if `mapAdminToManage` is true then the legacy `admin` permission will be mapped to the `manage` permission.

- The `admin` permission in ActiveMQ "Classic" relates to whether or not the broker will auto-create a destination if it doesn't exist and the user sends a message to it. Artemis automatically allows the automatic creation of a destination if the user has permission to send message to it. Therefore, the plugin will map the `admin` permission to the 6 aforementioned permissions in Artemis by default. If `mapAdminToManage` is true then the legacy `admin` permission will be mapped to the `manage` permission as well.

### 43.4. Secure Sockets Layer (SSL) Transport

When messaging clients are connected to servers, or servers are connected to other servers (e.g. via bridges) over an untrusted network then Apache ActiveMQ Artemis allows that traffic to be encrypted using the Secure Sockets Layer (SSL) transport.

For more information on configuring the SSL transport, please see [Configuring the Transport](#).

### 43.5. User credentials

Apache ActiveMQ Artemis ships with three security manager implementations:

- The flexible, pluggable `ActiveMQJAASSecurityManager` which supports any standard JAAS login module. Artemis ships with several login modules which will be discussed further down. This is the default security manager.

- The `ActiveMQBasicSecurityManager` which doesn't use JAAS and only supports auth via username & password credentials. It also supports adding, removing, and updating users via the management API. All user & role data is stored in the broker's bindings journal which means any changes made to a primary broker will be available on its backup.

- The legacy, deprecated `ActiveMQSecurityManagerImpl` that reads user credentials, i.e. user names,
passwords and role information from properties files on the classpath called `artemis-users.properties` and `artemis-roles.properties`.

43.5.1. JAAS Security Manager

When using the Java Authentication and Authorization Service (JAAS) much of the configuration depends on which login module is used. However, there are a few commonalities for every case. The first place to look is in `bootstrap.xml`. Here is an example using the `PropertiesLogin` JAAS login module which reads user, password, and role information from properties files:

```xml
<jaas-security domain="PropertiesLogin"/>
```

No matter what login module you’re using, you’ll need to specify it here in `bootstrap.xml`. The `domain` attribute here refers to the relevant login module entry in `login.config`. For example:

```java
PropertiesLogin {
    org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule required
debug=true
    org.apache.activemq.jaas.properties.user="artemis-users.properties"
    org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};
```

The `login.config` file is a standard JAAS configuration file. You can read more about this file on Oracle’s website. In short, the file defines:

- an alias for an entry (e.g. `PropertiesLogin`)
- the implementation class for the login module (e.g. `org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule`)
- a flag which indicates whether the success of the login module is `required`, `requisite`, `sufficient`, or `optional` (see more details on these flags in the JavaDoc)
- a list of configuration options specific to the login module implementation

By default, the location and name of `login.config` is specified on the Artemis command-line which is set by `etc/artemis.profile` on Linux and `etc\artemis.profile.cmd` on Windows.

Dual Authentication

The JAAS Security Manager also supports another configuration parameter - `certificate-domain`. This is useful when you want to authenticate clients connecting with SSL connections based on their SSL certificates (e.g. using the `CertificateLoginModule` discussed below) but you still want to authenticate clients connecting with non-SSL connections with, e.g., username and password. Here’s an example of what would go in `bootstrap.xml`:

```xml
<jaas-security domain="PropertiesLogin" certificate-domain="CertLogin"/>
```
And here's the corresponding `login.config`:

```java
PropertiesLogin {
    org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule required
debug=false
    org.apache.activemq.jaas.properties.user="artemis-users.properties"
    org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};

CertLogin {
    org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginModule
required
debug=true
    org.apache.activemq.jaas.textfiledn.user="cert-users.properties"
    org.apache.activemq.jaas.textfiledn.role="cert-roles.properties";
};
```

When the broker is configured this way then any client connecting with SSL and a client certificate will be authenticated using `CertLogin` and any client connecting without SSL will be authenticated using `PropertiesLogin`.

### 43.5.2. JAAS Login Modules

**GuestLoginModule**

Allows users without credentials (and, depending on how it is configured, possibly also users with invalid credentials) to access the broker. Normally, the guest login module is chained with another login module, such as a properties login module. It is implemented by `org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule`.

- **org.apache.activemq.jaas.guest.user**
  
  the user name to assign; default is "guest"

- **org.apache.activemq.jaas.guest.role**
  
  the role name to assign; default is "guests"

- **credentialsInvalidate**
  
  boolean flag; if true, reject login requests that include a password (i.e. guest login succeeds only when the user does not provide a password); default is false

- **debug**
  
  boolean flag; if true, enable debugging; this is used only for testing or debugging; normally, it should be set to false, or omitted; default is false

There are two basic use cases for the guest login module, as follows:

- Guests with no credentials or invalid credentials.
- Guests with no credentials only.
The following snippet shows how to configure a JAAS login entry for the use case where users with no credentials or invalid credentials are logged in as guests. In this example, the guest login module is used in combination with the properties login module.

```
activemq-domain {
    org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule sufficient
debug=true
    org.apache.activemq.jaas.properties.user="artemis-users.properties"
    org.apache.activemq.jaas.properties.role="artemis-roles.properties";

    org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule sufficient
debug=true
    org.apache.activemq.jaas.guest.user="anyone"
    org.apache.activemq.jaas.guest.role="restricted";
};
```

Depending on the user login data, authentication proceeds as follows:

- **User logs in with a valid password** — the properties login module successfully authenticates the user and returns immediately. The guest login module is not invoked.
- **User logs in with an invalid password** — the properties login module fails to authenticate the user, and authentication proceeds to the guest login module. The guest login module successfully authenticates the user and returns the guest principal.
- **User logs in with a blank password** — the properties login module fails to authenticate the user, and authentication proceeds to the guest login module. The guest login module successfully authenticates the user and returns the guest principal.

The following snippet shows how to configure a JAAS login entry for the use case where only those users with no credentials are logged in as guests. To support this use case, you must set the credentialsInvalidate option to true in the configuration of the guest login module. You should also note that, compared with the preceding example, the order of the login modules is reversed and the flag attached to the properties login module is changed to requisite.

```
activemq-guest-when-no-creds-only-domain {
    org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule sufficient
debug=true
    credentialsInvalidate=true
    org.apache.activemq.jaas.guest.user="guest"
    org.apache.activemq.jaas.guest.role="guests";

    org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule requisite
debug=true
    org.apache.activemq.jaas.properties.user="artemis-users.properties"
    org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};
```

Depending on the user login data, authentication proceeds as follows:
• User logs in with a valid password—the guest login module fails to authenticate the user (because the user has presented a password while the credentialsInvalidate option is enabled) and authentication proceeds to the properties login module. The properties login module successfully authenticates the user and returns.

• User logs in with an invalid password—the guest login module fails to authenticate the user and authentication proceeds to the properties login module. The properties login module also fails to authenticate the user. The net result is authentication failure.

• User logs in with a blank password—the guest login module successfully authenticates the user and returns immediately. The properties login module is not invoked.

PropertiesLoginModule

The JAAS properties login module provides a simple store of authentication data, where the relevant user data is stored in a pair of flat files. This is convenient for demonstrations and testing, but for an enterprise system, the integration with LDAP is preferable. It is implemented by org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule.

org.apache.activemq.jaas.properties.user
the path to the file which contains user and password properties

org.apache.activemq.jaas.properties.role
the path to the file which contains user and role properties

org.apache.activemq.jaas.properties.password.codec
the fully qualified class name of the password codec to use. See the password masking documentation for more details on how this works.

reload
boolean flag; whether or not to reload the properties files when a modification occurs; default is false

debug
boolean flag; if true, enable debugging; this is used only for testing or debugging; normally, it should be set to false, or omitted; default is false

In the context of the properties login module, the artemis-users.properties file consists of a list of properties of the form, UserName=Password. For example, to define the users system, user, and guest, you could create a file like the following:

```properties
system=manager
user=password
guest=password
```

Passwords in artemis-users.properties can be hashed. Such passwords should follow the syntax ENC(<hash>).

Hashed passwords can easily be added to artemis-users.properties using the user CLI command
from the Artemis instance. This command will not work from the Artemis home, and it will also not work unless the broker has been started.

```
./artemis user add --user-command-user guest --user-command-password guest --role admin
```

This will use the default codec to perform a "one-way" hash of the password and alter both the artemis-users.properties and artemis-roles.properties files with the specified values.

The artemis-roles.properties file consists of a list of properties of the form, Role=UserList, where UserList is a comma-separated list of users. For example, to define the roles admins, users, and guests, you could create a file like the following:

```
admins=system
users=system,user
guests=guest
```

As mentioned above, the Artemis command-line interface supports a command to add a user. Commands to list (one or all) users, remove a user, and reset a user's password and/or role(s) are also supported via the command-line interface as well as the normal management interfaces (e.g. JMX, web console, etc.).

**Warning**

Management and CLI operations to manipulate user & role data are only available when using the PropertiesLoginModule.

In general, using properties files and broker-centric user management for anything other than very basic use-cases is not recommended. The broker is designed to deal with messages. It's not in the business of managing users, although that functionality is provided at a limited level for convenience. LDAP is recommended for enterprise level production use-cases.

**LDAPLoginModule**

The LDAP login module enables you to perform authentication and authorization by checking the incoming credentials against user data stored in a central X.500 directory server. For systems that already have an X.500 directory server in place, this means that you can rapidly integrate ActiveMQ Artemis with the existing security database and user accounts can be managed using the X.500 system. It is implemented by org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule.

**initialContextFactory**

must always be set to com.sun.jndi.ldap.LdapCtxFactory
**connectionURL**

specify the location of the directory server using an ldap URL, ldap://Host:Port. You can optionally qualify this URL, by adding a forward slash, /, followed by the DN of a particular node in the directory tree. For example, ldap://ldapserver:10389/ou/system.

**authentication**

specifies the authentication method used when binding to the LDAP server. Can take either of the values, **simple** (username and password), **GSSAPI** (Kerberos SASL) or **none** (anonymous).

**connectionUsername**

the DN of the user that opens the connection to the directory server. For example, **uid=admin,ou=system**. Directory servers generally require clients to present username/password credentials in order to open a connection.

**connectionPassword**

the password that matches the DN from **connectionUsername**. In the directory server, in the DIT, the password is normally stored as a **userPassword** attribute in the corresponding directory entry.

**saslLoginConfigScope**

the scope in JAAS configuration (login.config) to use to obtain Kerberos initiator credentials when the authentication method is SASL **GSSAPI**. The default value is **broker-sasl-gssapi**.

**connectionProtocol**

currently, the only supported value is a blank string. In future, this option will allow you to select the Secure Socket Layer (SSL) for the connection to the directory server. This option must be set explicitly to an empty string, because it has no default value.

**connectionTimeout**

specifies the string representation of an integer representing the connection timeout in milliseconds. If the LDAP provider cannot establish a connection within that period, it aborts the connection attempt. The integer should be greater than zero. An integer less than or equal to zero means to use the network protocol's (i.e., TCP's) timeout value.

If **connectionTimeout** is not specified, the default is to wait for the connection to be established or until the underlying network times out.

When connection pooling has been requested for a connection, this property also determines the maximum wait time for a connection when all connections in the pool are in use and the maximum pool size has been reached. If the value of this property is less than or equal to zero under such circumstances, the provider will wait indefinitely for a connection to become available; otherwise, the provider will abort the wait when the maximum wait time has been exceeded. See **connectionPool** for more details.

**readTimeout**

specifies the string representation of an integer representing the read timeout in milliseconds for LDAP operations. If the LDAP provider cannot get a LDAP response within that period, it aborts the read attempt. The integer should be greater than zero. An integer less than or equal to zero means no read timeout is specified which is equivalent to waiting for the response
infinitely until it is received.

If `readTimeout` is not specified, the default is to wait for the response until it is received.

**userBase**

selects a particular subtree of the DIT to search for user entries. The subtree is specified by a DN, which specifies the base node of the subtree. For example, by setting this option to `ou=User,ou=ActiveMQ,ou=system`, the search for user entries is restricted to the subtree beneath the `ou=User,ou=ActiveMQ,ou=system` node.

**userSearchMatching**

specifies an LDAP search filter, which is applied to the subtree selected by `userBase`. Before passing to the LDAP search operation, the string value you provide here is subjected to string substitution, as implemented by the `java.text.MessageFormat` class. Essentially, this means that the special string, `{0}`, is substituted by the username, as extracted from the incoming client credentials. After substitution, the string is interpreted as an LDAP search filter, where the LDAP search filter syntax is defined by the IETF standard, RFC 2254. A short introduction to the search filter syntax is available from Oracle's JNDI tutorial, [Search Filters](https://docs.oracle.com/javase/8/docs/technotes/guides/jndi/howto/search.html).

For example, if this option is set to `(uid={0})` and the received username is `jdoe`, the search filter becomes `(uid=jdoe)` after string substitution. If the resulting search filter is applied to the subtree selected by the user base, `ou=User,ou=ActiveMQ,ou=system`, it would match the entry, `uid=jdoe,ou=User,ou=ActiveMQ,ou=system` (and possibly more deeply nested entries, depending on the specified search depth—see the `userSearchSubtree` option).

**userSearchSubtree**

specify the search depth for user entries, relative to the node specified by `userBase`. This option is a boolean. `false` indicates it will try to match one of the child entries of the `userBase` node (maps to `javax.naming.directory.SearchControls.ONELEVEL_SCOPE`). `true` indicates it will try to match any entry belonging to the subtree of the `userBase` node (maps to `javax.naming.directory.SearchControls.SUBTREE_SCOPE`).

**userRoleName**

specifies the name of the multi-valued attribute of the user entry that contains a list of role names for the user (where the role names are interpreted as group names by the broker's authorization plug-in). If you omit this option, no role names are extracted from the user entry.

**roleBase**

if you want to store role data directly in the directory server, you can use a combination of role options (`roleBase`, `roleSearchMatching`, `roleSearchSubtree`, and `roleName`) as an alternative to (or in addition to) specifying the `userRoleName` option. This option selects a particular subtree of the DIT to search for role/group entries. The subtree is specified by a DN, which specifies the base node of the subtree. For example, by setting this option to `ou=Group,ou=ActiveMQ,ou=system`, the search for role/group entries is restricted to the subtree beneath the `ou=Group,ou=ActiveMQ,ou=system` node.

**roleName**

specifies the attribute type of the role entry that contains the name of the role/group (e.g. C, O,
OU, etc.). If you omit this option the full DN of the role is used.

**roleSearchMatching**

specifies an LDAP search filter, which is applied to the subtree selected by `roleBase`. This works in a similar manner to the `userSearchMatching` option, except that it supports two substitution strings, as follows:

- `{0}` - substitutes the full DN of the matched user entry (that is, the result of the user search). For example, for the user, `jdoe`, the substituted string could be `uid=jdoe,ou=User,ou=ActiveMQ,ou=system`.

- `{1}` - substitutes the received username. For example, `jdoe`.

For example, if this option is set to `(member=uid={1})` and the received username is `jdoe`, the search filter becomes `(member=uid=jdoe)` after string substitution (assuming ApacheDS search filter syntax). If the resulting search filter is applied to the subtree selected by the role base, `ou=Group,ou=ActiveMQ,ou=system`, it matches all role entries that have a `member` attribute equal to `uid=jdoe` (the value of a `member` attribute is a DN).

+ This option must always be set to enable role searching because it has no default value. Leaving it unset disables role searching and the role information must come from `userRoleName`.

+ If you use OpenLDAP, the syntax of the search filter is `(member:=uid=jdoe)`.

**roleSearchSubtree**

specify the search depth for role entries, relative to the node specified by `roleBase`. This option can take boolean values, as follows:

- `false` (default) - try to match one of the child entries of the roleBase node (maps to `javax.naming.directory.SearchControls.ONELEVEL_SCOPE`).

- `true` — try to match any entry belonging to the subtree of the roleBase node (maps to `javax.naming.directory.SearchControls.SUBTREE_SCOPE`).

**authenticateUser**

boolean flag to disable authentication. Useful as an optimisation when this module is used just for role mapping of a Subject’s existing authenticated principals; default is `true`.

**referral**

specify how to handle referrals; valid values: `ignore`, `follow`, `throw`; default is `ignore`.

**ignorePartialResultException**

boolean flag for use when searching Active Directory (AD). AD servers don’t handle referrals automatically, which causes a `PartialResultException` to be thrown when referrals are encountered by a search, even if referral is set to `ignore`. Set to `true` to ignore these exceptions; default is `false`.

**expandRoles**

boolean indicating whether to enable the role expansion functionality or not; default false. If enabled, then roles within roles will be found. For example, role `A` is in role `B`. User `X` is in role `A`, which means user `X` is in role `B` by virtue of being in role `A`.
**expandRolesMatching**

specifies an LDAP search filter which is applied to the subtree selected by roleBase. Before passing to the LDAP search operation, the string value you provide here is subjected to string substitution, as implemented by the java.text.MessageFormat class. Essentially, this means that the special string, `{0}`, is substituted by the role name as extracted from the previous role search. This option must always be set to enable role expansion because it has no default value. Example value: `(member={0})`.

**noCacheExceptions**

comma separated list of class names or regular expressions to match exceptions which may be thrown during communication with the LDAP server; default is empty. Typically any failure to authenticate will be stored in the authentication cache so that the underlying security data store (e.g. LDAP) is spared any unnecessary traffic. For example, an application with the wrong password attempting to login multiple times in short order might adversely impact the LDAP server. However, in cases where the failure is, for example, due to a temporary network outage and the security-invalidatio-interval is relatively high then not caching such failures would be better. Users can enumerate any relevant exceptions which the cache should ignore (e.g. java.net.ConnectException). The name of the exception or the regular expression should match the root cause from the relevant stack-trace. Users can confirm the configured exceptions are being skipped by enabling debug logging for org.apache.activemq.artemis.core.security.impl.SecurityStoreImpl.

**debug**

boolean flag; if true, enable debugging; this is used only for testing or debugging; normally, it should be set to false, or omitted; default is false

Any additional configuration option not recognized by the LDAP login module itself is passed as-is to the underlying LDAP connection logic.

Add user entries under the node specified by the userBase option. When creating a new user entry in the directory, choose an object class that supports the userPassword attribute (for example, the person or inetOrgPerson object classes are typically suitable). After creating the user entry, add the userPassword attribute, to hold the user's password.

If you want to store role data in dedicated role entries (where each node represents a particular role), create a role entry as follows. Create a new child of the roleBase node, where the objectClass of the child is groupOfNames. Set the cn (or whatever attribute type is specified by roleName) of the new child node equal to the name of the role/group. Define a member attribute for each member of the role/group, setting the member value to the DN of the corresponding user (where the DN is specified either fully, uid=jdoe,ou=User,ou=ActiveMQ,ou=system, or partially, uid=jdoe).

If you want to add roles to user entries, you would need to customize the directory schema, by adding a suitable attribute type to the user entry's object class. The chosen attribute type must be capable of handling multiple values.

**CertificateLoginModule**

The JAAS certificate authentication login module must be used in combination with SSL and the clients must be configured with their own certificate. In this scenario, authentication is actually
performed during the SSL/TLS handshake, not directly by the JAAS certificate authentication plug-in. The role of the plug-in is as follows:

- To further constrain the set of acceptable users, because only the user DNs explicitly listed in the relevant properties file are eligible to be authenticated.
- To associate a list of groups with the received user identity, facilitating integration with the authorization feature.
- To require the presence of an incoming certificate (by default, the SSL/TLS layer is configured to treat the presence of a client certificate as optional).

The JAAS certificate login module stores a collection of certificate DNs in a pair of flat files. The files associate a username and a list of group IDs with each DN.

The certificate login module is implemented by the following class:

```
org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginModule
```

The following `CertLogin` login entry shows how to configure certificate login module in the `login.config` file:

```
CertLogin {
    org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginModule
    debug=true
    org.apache.activemq.jaas.textfiledn.user="users.properties"
    org.apache.activemq.jaas.textfiledn.role="roles.properties"
};
```

In the preceding example, the JAAS realm is configured to use a single `org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginModule` login module. The options supported by this login module are as follows:

**debug**

boolean flag; if true, enable debugging; this is used only for testing or debugging; normally, it should be set to `false`, or omitted; default is `false`

**org.apache.activemq.jaas.textfiledn.user**

specifies the location of the user properties file (relative to the directory containing the login configuration file).

**org.apache.activemq.jaas.textfiledn.role**

specifies the location of the role properties file (relative to the directory containing the login configuration file).

**reload**

boolean flag; whether or not to reload the properties files when a modification occurs; default is `false`
In the context of the certificate login module, the `users.properties` file consists of a list of properties of the form, `UserName=StringifiedSubjectDN` or `UserName=/SubjectDNRegExp/`. For example, to define the users, `system`, `user` and `guest` as well as a `hosts` user matching several DNs, you could create a file like the following:

```properties
system=CN=system,O=Progress,C=US
user=CN=humble user,O=Progress,C=US
guest=CN=anon,O=Progress,C=DE
hosts=/CN=host\d+.acme\..com,O=Acme,C=UK/
```

Note that the backslash character has to be escaped because it has a special treatment in properties files.

Each username is mapped to a subject DN, encoded as a string (where the string encoding is specified by RFC 2253). For example, the system username is mapped to the `CN=system,O=Progress,C=US` subject DN. When performing authentication, the plug-in extracts the subject DN from the received certificate, converts it to the standard string format, and compares it with the subject DNs in the `users.properties` file by testing for string equality. Consequently, you must be careful to ensure that the subject DNs appearing in the `users.properties` file are an exact match for the subject DNs extracted from the user certificates.

Technically, there is some residual ambiguity in the DN string format. For example, the `domainComponent` attribute could be represented in a string either as the string, `DC`, or as the OID, `0.9.2342.19200300.100.1.25`. Normally, you do not need to worry about this ambiguity. But it could potentially be a problem, if you changed the underlying implementation of the Java security layer.

The easiest way to obtain the subject DNs from the user certificates is by invoking the `keytool` utility to print the certificate contents. To print the contents of a certificate in a keystore, perform the following steps:

1. Export the certificate from the keystore file into a temporary file. For example, to export the certificate with alias `broker-localhost` from the `broker.ks` keystore file, enter the following command:

   ```bash
   keytool -export -file broker.export -alias broker-localhost -keystore broker.ks -storepass password
   ```

   After running this command, the exported certificate is in the file, `broker.export`.

2. Print out the contents of the exported certificate. For example, to print out the contents of `broker.export`, enter the following command:

   ```bash
   keytool -printcert -file broker.export
   ```

   Which should produce output similar to that shown here:
The string following Owner: gives the subject DN. The format used to enter the subject DN depends on your platform. The Owner: string above could be represented as either CN=localhost, OU=broker, O=Unknown, L=Unknown, ST=Unknown, C=Unknown or CN=localhost,OU=broker,O=Unknown,L=Unknown,ST=Unknown,C=Unknown.

The roles.properties file consists of a list of properties of the form, Role=UserList, where UserList is a comma-separated list of users. For example, to define the roles admins, users, and guests, you could create a file like the following:

```
admins=system
users=system,user
guests=guest
```

**SCRAMPropertiesLoginModule**

The SCRAM properties login module implements the SASL challenge response for the SCRAM-SHA mechanism. The data in the properties file reference via org.apache.activemq.jaas.properties.user needs to be generated by the login module it's self, as part of user registration. It contains proof of knowledge of passwords, rather than passwords themselves. For more usage detail refer to SCRAM-SHA SASL Mechanism.

```
amqp-sasl-scram {
    org.apache.activemq.artemis.spi.core.security.jaas.SCRAMPropertiesLoginModule
    required
        org.apache.activemq.jaas.properties.user="artemis-users.properties"
        org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};
```

**SCRAMLoginModule**

The SCRAM login module converts a valid SASL SCRAM-SHA Authenticated identity into a JAAS User Principal. This Principal can then be used for role mapping.

```
{
    org.apache.activemq.artemis.spi.core.security.jaas.SCRAMLLoginModule
};
```
**ExternalCertificateLoginModule**

The external certificate login module is used to propagate a validated TLS client certificate’s subjectDN into a JAAS UserPrincipal. This allows subsequent login modules to do role mapping for the TLS client certificate.

```
org.apache.activemq.artemis.spi.core.security.jaas.ExternalCertificateLoginModule required;
```

**PrincipalConversionLoginModule**

The principal conversion login module is used to convert an existing validated Principal into a JAAS UserPrincipal. The module is configured with a list of class names used to match existing Principals. If no UserPrincipal exists, the first matching Principal will be added as a UserPrincipal of the same Name.

```
org.apache.activemq.artemis.spi.core.security.jaas.PrincipalConversionLoginModule required
   principalClassList=org.apache.x.Principal,org.apache.y.Principal;
```

**Krb5LoginModule**

The Kerberos login module is used to propagate a validated SASL GSSAPI kerberos token identity into a validated JAAS UserPrincipal. This allows subsequent login modules to do role mapping for the kerberos identity.

```
org.apache.activemq.artemis.spi.core.security.jaas.Krb5LoginModule required;
```

The simplest way to make the login configuration available to JAAS is to add the directory containing the file, `login.config`, to your CLASSPATH.

**KubernetesLoginModule**

The Kubernetes login module enables you to perform authentication and authorization by validating the Bearer token against the Kubernetes API. The authentication is done by submitting a TokenReview request that the Kubernetes cluster validates. The response will tell whether the user is authenticated and the associated username and roles. It is implemented by `org.apache.activemq.artemis.spi.core.security.jaas.KubernetesLoginModule`.

**ignoreTokenReviewRoles**

  - when true, do not map roles from the TokenReview user groups. default false
org.apache.activemq.jaas.kubernetes.role
the optional path to the file which contains role mapping, useful when ignoreTokenReviewRoles=true

reload
boolean flag; whether or not to reload the properties file when a modification occurs; default is false

debug
boolean flag; if true, enable debugging; this is used only for testing or debugging; normally, it should be set to false, or omitted; default is false

The login module must be allowed to query the required Rest API. For that, it will use the available token under /var/run/secrets/kubernetes.io/serviceaccount/token. Besides, in order to trust the connection the client will use the ca.crt file existing in the same folder. These two files will be mounted in the container. The service account running the KubernetesLoginModule must be allowed to create::TokenReview. The system:auth-delegator role is typically use for that purpose.

The optional roles properties file consists of a list of properties of the form, Role=UserList, where UserList is a comma-separated list of users. For example, to define the roles admins, users, and guests, you could create a file like the following:

```
admins=system:serviceaccounts:example-ns:admin-sa
users=system:serviceaccounts:other-ns:test-sa
```

43.5.3. SCRAM-SHA SASL Mechanism

SCRAM (Salted Challenge Response Authentication Mechanism) is an authentication mechanism that can establish mutual authentication using passwords. Apache ActiveMQ Artemis supports SCRAM-SHA-256 and SCRAM-SHA-512 SASL mechanisms to provide authentication for AMQP connections.

The following properties of SCRAM make it safe to use SCRAM-SHA even on unencrypted connections:

- The passwords are not sent in the clear over the communication channel. The client is challenged to offer proof it knows the password of the authenticating user, and the server is challenged to offer proof it had the password to initialise its authentication store. Only the proof is exchanged.
- The server and client each generate a new challenge for each authentication exchange, making it resilient against replay attacks.

Configuring the server to use SCRAM-SHA

The desired SCRAM-SHA mechanisms must be enabled on the AMQP acceptor in broker.xml by adding them to the saslMechanisms list url parameter. In this example, SASL is restricted to only the SCRAM-SHA-256 mechanism:
Of note is the reference to the sasl login config scope `saslLoginConfigScope=amqp-sasl-scram` that holds the relevant SCRAM login module. The mechanism makes use of JAAS to complete the SASL exchanges.

An example configuration scope for `login.config` that will implement SCRAM-SHA-256 using property files, is as follows:

```
amqp-sasl-scram {
    org.apache.activemq.artemis.spi.core.security.jaas.SCRAMPropertiesLoginModule
    required
    org.apache.activemq.jaas.properties.user="artemis-users.properties"
    org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};
```

### Configuring a user with SCRAM-SHA data on the server

With SCRAM-SHA, the server's users properties file do not contain any passwords, instead they contain derivative data that can be used to respond to a challenge. The secure encoded form of the password must be generated using the main method of `org.apache.activemq.artemis.spi.core.security.jaas.SCRAMPropertiesLoginModule` from the artemis-server module and inserting the resulting lines into your `artemis-users.properties` file.

```
java -cp "<distro-lib-dir>/*
org.apache.activemq.artemis.spi.core.security.jaas.SCRAMPropertiesLoginModule
<username> <password> [<iterations>]
```

An sample of the output can be found in the amqp examples, examples/protocols/amqp/sasl-scram/src/main/resources/activemq/server0/artemis-users.properties

### 43.5.4. Kerberos Authentication

You must have the Kerberos infrastructure set up in your deployment environment before the server can accept Kerberos credentials. The server can acquire its Kerberos acceptor credentials by using JAAS and a Kerberos login module. The JDK provides the `Krb5LoginModule` which executes the necessary Kerberos protocol steps to authenticate and obtain Kerberos credentials.

#### GSSAPI SASL Mechanism

Using SASL over AMQP, Kerberos authentication is supported using the **GSSAPI** SASL mechanism. With SASL doing Kerberos authentication, TLS can be used to provide integrity and confidentiality to the communications channel in the normal way.

The **GSSAPI** SASL mechanism must be enabled on the AMQP acceptor in `broker.xml` by adding it to
the \texttt{saslMechanisms} list url parameter: \texttt{saslMechanisms=GSSAPI,PLAIN, etc}.

\begin{verbatim}
<acceptor name="amqp">
tcp://0.0.0.0:5672?protocols=AMQP;saslMechanisms=GSSAPI</acceptor>
\end{verbatim}

The GSSAPI mechanism implementation on the server will use a JAAS configuration scope named \texttt{amqp-sasl-gssapi} to obtain its Kerberos acceptor credentials. An alternative configuration scope can be specified on the AMQP acceptor using the url parameter: \texttt{saslLoginConfigScope=<some other scope>}. An example configuration scope for \texttt{login.config} that will pick up a Kerberos keyTab for the Kerberos acceptor Principal \texttt{amqp/localhost} is as follows:

\begin{verbatim}
 amqp-sasl-gssapi {
  com.sun.security.auth.module.Krb5LoginModule required
  isInitiator=false
  storeKey=true
  useKeyTab=true
  principal="amqp/localhost"
  debug=true;
};
\end{verbatim}

\section*{43.5.5. Role Mapping}

On the server, a Kerberos or SCRAM-SHA JAAS authenticated Principal must be added to the Subject's principal set as an Apache ActiveMQ Artemis UserPrincipal using the corresponding Apache ActiveMQ Artemis \texttt{Krb5LoginModule} or \texttt{SCRAMLoginModule} login modules. They are separate to allow conversion and role mapping to be as restrictive or permissive as desired.

The \texttt{PropertiesLoginModule} or \texttt{LDAPLoginModule} can then be used to map the authenticated Principal to an Apache ActiveMQ Artemis \texttt{Role}. Note that in the case of Kerberos, the Peer Principal does not exist as an Apache ActiveMQ Artemis user, only as a role member.

In the following example, any existing Kerberos authenticated peer will convert to an Apache ActiveMQ Artemis user principal and will have role mapping applied by the \texttt{LDAPLoginModule} as appropriate.

\begin{verbatim}
 activemq {
  org.apache.activemq.artemis.spi.core.security.jaas.Krb5LoginModule required
  ;
  org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule optional
  initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
  connectionURL="ldap://localhost:1024"
  authentication=GSSAPI
  saslLoginConfigScope=broker-sasl-gssapi
  connectionProtocol=s
  userBase="ou=users,dc=example,dc=com"
};
\end{verbatim}
43.5.6. Basic Security Manager

As the name suggests, the ActiveMQBasicSecurityManager is basic. It is not pluggable like the JAAS security manager and it only supports authentication via username and password credentials. Furthermore, the Hawtio-based web console requires JAAS. Therefore you will still need to configure a login.config if you plan on using the web console. However, this security manager may still may have a couple of advantages depending on your use-case.

All user & role data is stored in the bindings journal (or bindings table if using JDBC). The advantage here is that in a primary/backup use-case any user management performed on the primary broker will be reflected on the backup upon failover.

 Typically LDAP would be employed for this kind of use-case, but not everyone wants or is able to administer an independent LDAP server. One significant benefit of LDAP is that user data can be shared between multiple active brokers. However, this is not possible with the ActiveMQBasicSecurityManager or, in fact, any other configuration potentially available out of the box. Nevertheless, if you just want to share user data between a single live/backup pair then the basic security manager may be a good fit for you.

User management is provided by the broker's management API. This includes the ability to add, list, update, and remove users & roles. As with all management functions, this is available via JMX, management messages, HTTP (via Jolokia), web console, etc. These functions are also available from the ActiveMQ Artemis command-line interface. Having the broker store this data directly means that it must be running in order to manage users. There is no way to modify the bindings data manually.

To be clear, any management access via HTTP (e.g. web console or Jolokia) will go through Hawtio JAAS. MBean access via JConsole or other remote JMX tool will go through the basic security manager. Management messages will also go through the basic security manager.

Configuration

The configuration for the ActiveMQBasicSecurityManager happens in bootstrap.xml just like it does for all security manager implementations. Start by removing <jaas-security /> section and add <security-manager /> configuration as described below.

The ActiveMQBasicSecurityManager requires some special configuration for the following reasons:

- the bindings data which holds the user & role data cannot be modified manually
• the broker must be running to manage users
• the broker often needs to be secured from first boot

If, for example, the broker was configured to use the ActiveMQBasicSecurityManager and was started from scratch then no clients would be able to connect because there would be no users & roles configured. However, in order to configure users & roles one would need to use the management API which would require the proper credentials. It's a catch-22 problem. Therefore, it is essential to configure "bootstrap" credentials that will be automatically created when the broker starts. There are properties to define either:

• a single user whose credentials can then be used to add other users
• properties files from which to load users & roles in bulk

Here's an example of the single bootstrap user configuration:

```xml
<broker xmlns="http://activemq.apache.org/schema">
  <security-manager class-name="org.apache.activemq.artemis.spi.core.security.ActiveMQBasicSecurityManager">
    <property key="bootstrapUser" value="myUser"/>
    <property key="bootstrapPassword" value="myPass"/>
    <property key="bootstrapRole" value="myRole"/>
  </security-manager>
  ...
</broker>
```

**bootstrapUser**

The name of the bootstrap user.

**bootstrapPassword**

The password for the bootstrap user; supports masking.

**bootstrapRole**

The role of the bootstrap user.

If your use-case requires multiple users to be available when the broker starts then you can use a configuration like this:

```xml
<broker xmlns="http://activemq.apache.org/schema">
  <security-manager class-name="org.apache.activemq.artemis.spi.core.security.ActiveMQBasicSecurityManager">
    <property key="bootstrapUserFile" value="artemis-users.properties"/>
    <property key="bootstrapRoleFile" value="artemis-roles.properties"/>
  </security-manager>
  ...
</broker>
```
bootstrapUserFile

The name of the file from which to load users. This is a properties file formatted exactly the same as the user properties file used by the PropertiesLoginModule. This file should be on the broker's classpath (e.g. in the etc directory).

bootstrapRoleFile

The role of the bootstrap user. This is a properties file formatted exactly the same as the role properties file used by the PropertiesLoginModule. This file should be on the broker's classpath (e.g. in the etc directory).

Regardless of whether you configure a single bootstrap user or load many users from properties files, any user with which additional users are created should be in a role with the appropriate permissions on the activemq.management address. For example if you've specified a bootstrapUser then the bootstrapRole will need the following permissions:

- createNonDurableQueue
- createAddress
- consume
- manage
- send

For example:

```
<security-setting match="activemq.management.#">
  <permission type="createNonDurableQueue" roles="myRole"/>
  <permission type="createAddress" roles="myRole"/>
  <permission type="consume" roles="myRole"/>
  <permission type="manage" roles="myRole"/>
  <permission type="send" roles="myRole"/>
</security-setting>
```

Any bootstrap credentials will be reset whenever you start the broker no matter what changes may have been made to them at runtime previously, so depending on your use-case you should decide if you want to leave bootstrap configuration permanent or if you want to remove it after initial configuration.

### 43.6. Mapping external roles

Roles from external authentication providers (i.e. LDAP) can be mapped to internally used roles. The is done through role-mapping entries in the security-settings block:

```
<security-settings>
  [...]
</security-settings>
```
Role mapping is additive. That means the user will keep the original role(s) as well as the newly assigned role(s).

This role mapping only affects the roles which are used to authorize queue access through the configured acceptors. It can not be used to map the role required to access the web console.

43.7. SASL

AMQP supports SASL. The following mechanisms are supported: PLAIN, EXTERNAL, ANONYMOUS, GSSAPI, SCRAM-SHA-256, SCRAM-SHA-512. The published list can be constrained via the amqp acceptor `saslMechanisms` property.

EXTERNAL will only be chosen if a subject is available from the TLS client certificate.

43.8. Changing the username/password for clustering

In order for cluster connections to work correctly, each node in the cluster must make connections to the other nodes. The username/password they use for this should always be changed from the installation default to prevent a security risk.

Please see Management for instructions on how to do this.

43.9. Securing the console

Artemis comes with a web console that allows user to browse Artemis documentation via an embedded server. By default the web access is plain HTTP. It is configured in `bootstrap.xml`:

```xml
<web path="web">
  <binding uri="http://localhost:8161">
    <app url="console" war="console.war"/>
  </binding>
</web>
```

Alternatively you can edit the above configuration to enable secure access using HTTPS protocol. e.g.:

```xml
<web path="web">
  <binding uri="https://localhost:8443">
    keyStorePath="${artemis.instance}/etc/keystore.jks"
  </binding>
</web>
```
As shown in the example, to enable https the first thing to do is config the bind to be an https url. In addition, you will have to configure a few extra properties described as below.

**keyStorePath**

The path of the key store file.

**keyStorePassword**

The key store's password.

**clientAuth**

The boolean flag indicates whether or not client authentication is required. Default is false.

**trustStorePath**

The path of the trust store file. This is needed only if clientAuth is true.

**trustStorePassword**

The trust store's password.

### 43.9.1. Config access using client certificates

The web console supports authentication with client certificates, see the following steps:

- Add the certificate login module to the login.config file, i.e.

  ```java
  activemq-cert {
  org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginModule
  required
  debug=true
  org.apache.activemq.jaas.textfiledn.user="cert-users.properties"
  org.apache.activemq.jaas.textfiledn.role="cert-roles.properties";
  }
  ```

- Change the hawtio realm to match the realm defined in the login.config file for the certificate login module. This is configured in the artemis.profile via the system property -Dhawtio.role=activemq-cert.

- Create a key pair for the client and import the public key in a truststore file.

  ```bash
  keytool -storetype pkcs12 -keystore client-keystore.p12 -storepass securepass
  -keypass securepass -alias client -genkey -keyalg "RSA" -keysize 2048 -dname
  "CN=ActiveMQ Artemis Client, OU=Artemis, O=ActiveMQ, L=AMQ, S=AMQ, C=AMQ" -ext
  bc=ca:false -ext eku=cA
  keytool -storetype pkcs12 -keystore client-keystore.p12 -storepass securepass
  ```
Enable secure access using HTTPS protocol with client authentication, use the truststore file created in the previous step to set the `trustStorePath` and `trustStorePassword`:

```xml
<web path="web">
  <binding uri="https://localhost:8443">
    <app url="jolokia" war="jolokia-war-1.3.5.war"></app>
  </binding>
</web>
```

Use the private key created in the previous step to set up your client, i.e. if the client app is a browser install the private key in the browser.

### 43.10. Controlling JMS ObjectMessage deserialization

Artemis provides a simple class filtering mechanism with which a user can specify which packages are to be trusted and which are not. Objects whose classes are from trusted packages can be deserialized without problem, whereas those from 'not trusted' packages will be denied deserialization.

Artemis keeps a **deny list** to keep track of packages that are not trusted and a **allow list** for trusted packages. By default both lists are empty, meaning any serializable object is allowed to be deserialized. If an object whose class matches one of the packages in deny list, it is not allowed to be deserialized. If it matches one in the allow list the object can be deserialized. If a package appears in both deny list and allow list, the one in deny list takes precedence. If a class neither matches with deny list nor with the allow list, the class deserialization will be denied unless the allow list is empty (meaning the user doesn’t specify the allow list at all).

A class is considered as a 'match' if:

- its full name exactly matches one of the entries in the list.
- its package matches one of the entries in the list or is a sub-package of one of the entries.

For example, if a class full name is "org.apache.pkg1.Class1", some matching entries could be:

- `org.apache.pkg1` - exact package match.
- `org.apache` — sub package match.
A * means 'match-all' in a deny or allow list.

43.10.1. Config via Connection Factories

To specify the allow and deny lists one can use the URL parameters deserializationDenyList and deserializationAllowList. For example, using JMS:

```java
ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory("vm://0?deserializationDenyList=org.apache.pkg1,org.some.pkg2");
```

The above statement creates a factory that has a deny list contains two forbidden packages, "org.apache.pkg1" and "org.some.pkg2", separated by a comma.

43.10.2. Config via system properties

There are two system properties available for specifying deny list and allow list:

- `org.apache.activemq.artemis.jms.deserialization.allowlist` comma separated list of entries for the allow list.
- `org.apache.activemq.artemis.jms.deserialization.denylist` comma separated list of entries for the deny list.

Once defined, all JMS object message deserialization in the VM is subject to checks against the two lists. However if you create a ConnectionFactory and set a new set of deny/allow lists on it, the new values will override the system properties.

43.10.3. Config for resource adapters

Message beans using a JMS resource adapter to receive messages can also control their object deserialization via properly configuring relevant properties for their resource adapters. There are two properties that you can configure with connection factories in a resource adapter:

- `deserializationDenyList` comma separated values for deny list
- `deserializationAllowList` comma separated values for allow list

These properties, once specified, are eventually set on the corresponding internal factories.

43.11. Masking Passwords

For details about masking passwords in broker.xml please see the Masking Passwords chapter.
43.12. Custom Security Manager

The underpinnings of the broker's security implementation can be changed if so desired. The broker uses a component called a "security manager" to implement the actual authentication and authorization checks. By default, the broker uses org.apache.activemq.artemis.spi.core.security.ActiveMQJAASSecurityManager to provide JAAS integration, but users can provide their own implementation of org.apache.activemq.artemis.spi.core.security.ActiveMQSecurityManager and configure it in bootstrap.xml using the security-manager element, e.g.:

```xml
<broker xmlns="http://activemq.apache.org/schema">
  <security-manager class-name="com.foo.MySecurityManager">
    <property key="myKey1" value="myValue1"/>
    <property key="myKey2" value="myValue2"/>
  </security-manager>
  ...
</broker>
```

The security-manager example demonstrates how to do this is more detail.

43.13. Per-Acceptor Security Domains

It's possible to override the broker's JAAS security domain by specifying a security domain on an individual acceptor. Simply use the securityDomain parameter and indicate which domain from your login.config to use, e.g.:

```xml
<acceptor name="myAcceptor">
  tcp://127.0.0.1:61616?securityDomain=mySecurityDomain</acceptor>
```

Any client connecting to this acceptor will be have security enforced using mySecurityDomain.
Chapter 44. Masking Passwords

By default all passwords in Apache ActiveMQ Artemis server’s configuration files are in plain text form. This usually poses no security issues as those files should be well protected from unauthorized accessing. However, in some circumstances a user doesn’t want to expose its passwords to more eyes than necessary.

Apache ActiveMQ Artemis can be configured to use 'masked' passwords in its configuration files. A masked password is an obscure string representation of a real password. To mask a password a user will use an 'codec'. The codec takes in the real password and outputs the masked version. A user can then replace the real password in the configuration files with the new masked password. When Apache ActiveMQ Artemis loads a masked password it uses the codec to decode it back into the real password.

Apache ActiveMQ Artemis provides a default codec. Optionally users can use or implement their own codec for masking the passwords.

In general, a masked password can be identified using one of two ways. The first one is the $ENC()$ syntax, i.e. any string value wrapped in $ENC()$ is to be treated as a masked password. For example $ENC(xyz)$

The above indicates that the password is masked and the masked value is $xyz$.

The $ENC()$ syntax is the preferred way of masking a password and is universally supported in every password configuration in Artemis.

The other, legacy way is to use a $mask-password$ attribute to tell that a password in a configuration file should be treated as 'masked'. For example:

```
<mask-password>true</mask-password>
<cluster-password>xyz</cluster-password>
```

This method is now deprecated and exists only to maintain backward-compatibility. Newer configurations may not support it.

44.1. Generating a Masked Password

To mask a password use the $mask$ command from the $bin$ directory of your Artemis instance. This command will not work from the Artemis home.

The $mask$ command uses the default codec unless a custom codec is defined in $broker.xml$ and the $--password-codec$ option is true. Here’s a simple example:

```
./artemis mask <plaintextPassword>
```

You’ll get something like:
Just copy `32c6f67dae6cd61b0a7ad1702033aa81e6b2a760123f4360` and replace your plaintext password with it using the `ENC()` syntax, e.g. `ENC(32c6f67dae6cd61b0a7ad1702033aa81e6b2a760123f4360)`.

You can also use the `--key` parameter with the default codec. Read more about the default codec for further details about this parameter.

This process works for passwords in:

- `broker.xml`
- `login.config`
- `bootstrap.xml`
- `management.xml`

This process does **not** work for passwords in:

- `artemis-users.properties`

Masked passwords for `artemis-users.properties` can be generated using the `mask` command using the `--hash` command-line option. However, this is also possible using the set of tools provided by the `user` command described below.

### 44.2. Masking Configuration

Besides supporting the `ENC()` syntax, the server configuration file (i.e. `broker.xml`) has a property that defines the default masking behaviors over the entire file scope.

`mask-password`: this boolean type property indicates if a password should be masked or not. Set it to `true` if you want your passwords masked. The default value is `false`. As noted above, this configuration parameter is deprecated in favor of the `ENC()` syntax.

`password-codec`: this string type property identifies the name of the class which will be used to decode the masked password within the broker. If not specified then the default `org.apache.activemq.artemis.utils.DefaultSensitiveStringCodec` will be used. Read more about using a custom codec.

#### 44.2.1. `artemis-users.properties`

Apache ActiveMQ Artemis' default JAAS security manager uses plain properties files where the user passwords are specified in a hashed form by default. Note, the passwords are technically *hashed* rather than masked in this context. The default `PropertiesLoginModule` will not decode the passwords in `artemis-users.properties` but will instead hash the input and compare the two hashed values for password verification.

Use the following command from the CLI of the Artemis instance you wish to add the user/password to. This command will not work from the Artemis home used to create the instance,
and it will also not work unless the broker has been started. For example:

```
./artemis user add --user-command-user guest --user-command-password guest --role admin
```

This will use the default codec to perform a "one-way" hash of the password and alter both the `artemis-users.properties` and `artemis-roles.properties` files with the specified values.

Passwords in `artemis-users.properties` are automatically detected as hashed or not by looking for the syntax `ENC(<hash>)`. The `mask-password` parameter does not need to be `true` to use hashed passwords here.

Management and CLI operations to manipulate user & role data are only available when using the `PropertiesLoginModule`.

In general, using properties files and broker-centric user management for anything other than very basic use-cases is not recommended. The broker is designed to deal with messages. It's not in the business of managing users, although that functionality is provided at a limited level for convenience. LDAP is recommended for enterprise level production use-cases.

### 44.2.2. `cluster-password`

If it is specified in `ENC()` syntax it will be treated as masked, or if `mask-password` is `true` the `cluster-password` will be treated as masked.

### 44.2.3. Connectors & Acceptors

In broker.xml `connector` and `acceptor` configurations sometimes needs to specify passwords. For example, if a user wants to use an `acceptor` with `sslEnabled=true` it can specify `keyStorePassword` and `trustStorePassword`. Because Acceptors and Connectors are pluggable implementations, each transport will have different password masking needs.

The preferred way is simply to use the `ENC()` syntax.

If using the legacy `mask-password` and `password-codec` values then when a `connector` or `acceptor` is initialised, Apache ActiveMQ Artemis will add these values to the parameters using the keys `activemq.usemaskedpassword` and `activemq.passwordcodec` respectively. The Netty and InVM implementations will use these as needed and any other implementations will have access to these to use if they so wish.

### 44.2.4. Core Bridges

Core Bridges are configured in the server configuration file and so the masking of its `password` properties follows the same rules as that of `cluster-password`. It supports `ENC()` syntax.

For using `mask-password` property, the following table summarizes the relations among the above-mentioned properties.
<table>
<thead>
<tr>
<th>mask-password</th>
<th>cluster-password</th>
<th>acceptor/connector passwords</th>
<th>bridge password</th>
</tr>
</thead>
<tbody>
<tr>
<td>absent</td>
<td>plain text</td>
<td>plain text</td>
<td>plain text</td>
</tr>
<tr>
<td>false</td>
<td>plain text</td>
<td>plain text</td>
<td>plain text</td>
</tr>
<tr>
<td>true</td>
<td>masked</td>
<td>masked</td>
<td>masked</td>
</tr>
</tbody>
</table>

It is recommended that you use the `ENC()` syntax for new applications/deployments.

**Examples**

In the following examples if related attributed or properties are absent, it means they are not specified in the configure file.

- **Unmasked**

  ```xml
  <cluster-password>bbc</cluster-password>
  ```

  This indicates the cluster password is a plain text value `bbc`.

- **Masked 1**

  ```xml
  <cluster-password>ENC(80cf731af62c290)</cluster-password>
  ```

  This indicates the cluster password is a masked value `80cf731af62c290`.

- **Masked 2**

  ```xml
  <mask-password>true</mask-password>
  <cluster-password>80cf731af62c290</cluster-password>
  ```

  This indicates the cluster password is a masked value and Apache ActiveMQ Artemis will use the default codec to decode it. All other passwords in the configuration file, Connectors, Acceptors and Bridges, will also use masked passwords.

**44.2.5. bootstrap.xml**

The broker embeds a web-server for hosting some web applications such as a management console. It is configured in `bootstrap.xml` as a web component. The web server can be secured using the `https` protocol, and it can be configured with a keystore password and/or truststore password which by default are specified in plain text forms.

To mask these passwords you need to use `ENC()` syntax. The `mask-password` boolean is not supported here.

You can also set the `passwordCodec` attribute if you want to use a password codec other than the
default one. For example

```xml
<web path="web">
  <binding uri="https://localhost:8443"
           keyStorePassword="ENC(-5a2376c61c668aaf)"
           trustStorePassword="ENC(3d617352d12839eb71208edf41d66b34)">
    <app url="activemq-branding" war="activemq-branding.war"/>
  </binding>
</web>
```

### 44.2.6. management.xml

The broker embeds a JMX connector which is used for management. The connector can be secured using SSL and it can be configured with a keystore password and/or truststore password which by default are specified in plain text forms.

To mask these passwords you need to use `ENC()` syntax. The `mask-password` boolean is not supported here.

You can also set the `password-codec` attribute if you want to use a password codec other than the default one. For example

```xml
<connector
    connector-port="1099"
    connector-host="localhost"
    secured="true"
    key-store-path="myKeystore.jks"
    key-store-password="ENC(3a34fd21b82bf2a822fa49a8d8fa115d"
    trust-store-path="myTruststore.jks"
    trust-store-password="ENC(3a34fd21b82bf2a822fa49a8d8fa115d")/>
```

With this configuration, both passwords in ra.xml and all of its MDBs will have to be in masked form.

### 44.2.7. PropertiesLoginModule

Artemis supports Properties login module to be configured in JAAS configuration file (default name is `login.config`). By default, the passwords of the users are in plain text or masked with the default codec.

To use a custom codec class, set the `org.apache.activemq.jaas.properties.password.codec` property to the class name e.g. to use the `com.example.MySensitiveDataCodecImpl` codec class:

```java
PropertiesLoginWithPasswordCodec {
    org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule required
debug=true
org.apache.activemq.jaas.properties.user="users.properties"
}
```
44.2.8. LDAPLoginModule

Artemis supports LDAP login modules to be configured in JAAS configuration file (default name is `login.config`). When connecting to an LDAP server usually you need to supply a connection password in the config file. By default this password is in plain text form.

To mask it you need to configure the passwords in your login module using `ENC()` syntax. To specify a codec using the following property:

```
passwordCodec - the password codec class name. (the default codec will be used if it is absent)
```

For example:

```java
LDAPLoginExternalPasswordCodec {
    org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule required
    debug=true
    initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
    connectionURL="ldap://localhost:1024"
    connectionUsername="uid=admin,ou=system"
    connectionPassword="ENC(-170b9ef34d79ed12)"
    passwordCodec="org.apache.activemq.artemis.utils.DefaultSensitiveStringCodec;key=hello world"
    connectionProtocol=s
    authentication=simple
    userBase="ou=system"
    userSearchMatching="(uid={0})"
    userSearchSubtree=false
    roleBase="ou=system"
    roleName=dummyRoleName
    roleSearchMatching="(uid={1})"
    roleSearchSubtree=false
};
```

44.2.9. JCA Resource Adapter

Both ra.xml and MDB activation configuration have a `password` property that can be masked preferably using `ENC()` syntax.

Alternatively it can use an optional attribute in ra.xml to indicate that a password is masked:
**UseMaskedPassword**

If setting to "true" the passwords are masked. Default is false.

There is another property in ra.xml that can specify a codec:

**PasswordCodec**

Class name and its parameters for the codec used to decode the masked password. Ignored if UseMaskedPassword is false. The format of this property is a full qualified class name optionally followed by key/value pairs. It is the same format as that for JMS Bridges. Example:

Example 1 Using the ENC() syntax:

```xml
<config-property>
  <config-property-name>password</config-property-name>
  <config-property-type>String</config-property-type>
  <config-property-value>ENC(80cf731af62c290)</config-property-value>
</config-property>
<config-property>
  <config-property-name>PasswordCodec</config-property-name>
  <config-property-type>java.lang.String</config-property-type>
  <config-property-value>com.foo.ACodec;key=helloworld</config-property-value>
</config-property>
```

Example 2 Using the "UseMaskedPassword" property:

```xml
<config-property>
  <config-property-name>UseMaskedPassword</config-property-name>
  <config-property-type>boolean</config-property-type>
  <config-property-value>true</config-property-value>
</config-property>
<config-property>
  <config-property-name>password</config-property-name>
  <config-property-type>String</config-property-type>
  <config-property-value>80cf731af62c290</config-property-value>
</config-property>
<config-property>
  <config-property-name>PasswordCodec</config-property-name>
  <config-property-type>java.lang.String</config-property-type>
  <config-property-value>com.foo.ACodec;key=helloworld</config-property-value>
</config-property>
```

### 44.3. Choosing a codec for password masking

As described in the previous sections, all password masking requires a codec. A codec uses an algorithm to convert a masked password into its original clear text form in order to be used in various security operations. The algorithm used for decoding must match that for encoding. Otherwise the decoding may not be successful.
For user's convenience Apache ActiveMQ Artemis provides a default codec. However, a user can implement their own if they wish.

### 44.3.1. The Default Codec

Whenever no codec is specified in the configuration, the default codec is used. The class name for the default codec is `org.apache.activemq.artemis.utils.DefaultSensitiveStringCodec`. It has hashing, encoding, and decoding capabilities. It uses `java.crypto.Cipher` utilities to hash or encode a plaintext password and also to decode a masked string using the same algorithm and "key."

The "key" used here is important since the same key **must** be used to both mask and unmask the password. The key is just a string of characters which the codec feeds to the underlying algorithm. There is a default key in `org.apache.activemq.artemis.utils.DefaultSensitiveStringCodec`, but using the default key leaves open the possibility that nefarious actors could also use that key to unmask the password(s). Therefore, it is possible to supply your own key, and there are a few ways to do this.

1. Specify the key in the codec configuration using the `key=value` syntax. Depending on which password you're trying to mask the configuration specifics will differ slightly, but this can be done, for example, in `broker.xml` with `password-codec`:

   ```xml
   <password-codec>
   org.apache.activemq.artemis.utils.DefaultSensitiveStringCodec;key=myKey</password-codec>
   ```

   Similar configurations are possible in any file that supports password masking, e.g. `bootstrap.xml`, `login.config`, `management.xml`, etc. The main drawback with this method is that the key will be stored in plain-text in the configuration file(s).

2. Set the system property `-Dartemis.default.sensitive.string.codec.key=myKey`

3. Set the environment property `ARTEMIS_DEFAULT_SENSITIVE_STRING_CODEC_KEY`. The benefit of using this method is that the key is more obscure since it will not exist in any configuration file. It can be set immediately before the broker starts and then cleared from the environment immediately after the broker finishes starting.

### 44.3.2. Using a custom codec

It is possible to use a custom codec rather than the built-in one. Simply make sure the codec is in Apache ActiveMQ Artemis's classpath. The custom codec can also be service loaded rather than class loaded, if the codec's service provider is installed in the classpath. Then configure the server to use it as follows:

```xml
<password-codec>com.foo.SomeCodec;key1=value1;key2=value2</password-codec>
```

If your codec needs params passed to it you can do this via key/value pairs when configuring. For instance if your codec needs a "key-location" parameter, you can define like so:
Then configure your cluster-password like this:

```
<cluster-password>ENC(masked_password)</cluster-password>
```

When Apache ActiveMQ Artemis reads the cluster-password it will initialize the `NewCodec` and use it to decode "mask_password". It also process all passwords using the new defined codec.

### Implementing Custom Codecs

To use a different codec than the built-in one, you either pick one from existing libraries or you implement it yourself. All codecs must implement the `org.apache.activemq.artemis.utils.SensitiveDataCodec<String>` interface. So a new codec would be defined like

```java
public class MyCodec implements SensitiveDataCodec<String> {
  @Override
  public String decode(Object mask) throws Exception {
    // Decode the mask into clear text password.
    return "the password";
  }

  @Override
  public String encode(Object secret) throws Exception {
    // Mask the clear text password.
    return "the masked password";
  }

  @Override
  public void init(Map<String, String> params) {
    // Initialization done here. It is called right after the codec has been created.
  }

  @Override
  public boolean verify(char[] value, String encodedValue) {
    // Return true if the value matches the encodedValue.
    return checkValueMatchesEncoding(value, encodedValue);
  }
}
```

Last but not least, once you get your own codec please add it to the classpath otherwise the broker will fail to load it!
Chapter 45. Resource Limits

Sometimes it’s helpful to set particular limits on what certain users can do beyond the normal security settings related to authorization and authentication. For example, limiting how many connections a user can create or how many queues a user can create. This chapter will explain how to configure such limits.

45.1. Configuring Limits Via Resource Limit Limit Settings

Here is an example of the XML used to set resource limits:

```xml
<resource-limit-settings>
  <resource-limit-setting match="myUser">
    <max-connections>5</max-connections>
    <max-queues>3</max-queues>
  </resource-limit-setting>
</resource-limit-settings>
```

Unlike the `match` from `address-setting`, this `match` does not use any wild-card syntax. It’s a simple 1:1 mapping of the limits to a `user`.

**max-connections**

How many connections the matched user can make to the broker. The default is -1 which means there is no limit.

**max-queues**

How many queues the matched user can create. The default is -1 which means there is no limit.
Chapter 46. Performance Tuning

In this chapter we’ll discuss how to tune Apache ActiveMQ Artemis for optimum performance.

46.1. Tuning persistence

- To get the best performance from Apache ActiveMQ Artemis whilst using persistent messages it is recommended that the file store is used. Apache ActiveMQ Artemis also supports JDBC persistence, but there is a performance cost when persisting to a database vs local disk.

- Put the message journal on its own physical volume. If the disk is shared with other processes e.g. transaction co-ordinator, database or other journals which are also reading and writing from it, then this may greatly reduce performance since the disk head may be skipping all over the place between the different files. One of the advantages of an append only journal is that disk head movement is minimised - this advantage is destroyed if the disk is shared. If you're using paging or large messages make sure they're ideally put on separate volumes too.

- Minimum number of journal files. Set `journal-min-files` to a number of files that would fit your average sustainable rate. This number represents the lower threshold of the journal file pool.

- To set the upper threshold of the journal file pool. (`journal-min-files` being the lower threshold). Set `journal-pool-files` to a number that represents something near your maximum expected load. The journal will spill over the pool should it need to, but will shrink back to the upper threshold, when possible. This allows reuse of files, without taking up more disk space than required. If you see new files being created on the journal data directory too often, i.e. lots of data is being persisted, you need to increase the journal-pool-size, this way the journal would reuse more files instead of creating new data files, increasing performance.

- Journal file size. The journal file size should be aligned to the capacity of a cylinder on the disk. The default value 10MiB should be enough on most systems.

- Use `ASYNCIO` journal. If using Linux, try to keep your journal type as `ASYNCIO`. `ASYNCIO` will scale better than Java NIO.

- Tune `journal-buffer-timeout`. The timeout can be increased to increase throughput at the expense of latency.

- If you're running `ASYNCIO` you might be able to get some better performance by increasing `journal-max-io`. DO NOT change this parameter if you are running NIO.

- If you are 100% sure you don’t need power failure durability guarantees, disable `journal-data-sync` and use `NIO` or `MAPPED` journal: you’ll benefit a huge performance boost on writes with process failure durability guarantees.

46.2. Tuning JMS

There are a few areas where some tweaks can be done if you are using the JMS API

- Disable message id. Use the `setDisableMessageID()` method on the `MessageProducer` class to disable message ids if you don't need them. This decreases the size of the message and also avoids the overhead of creating a unique ID.
• Disable message timestamp. Use the `setDisableMessageTimeStamp()` method on the `MessageProducer` class to disable message timestamps if you don't need them.

• Avoid `ObjectMessage`. `ObjectMessage` is convenient but it comes at a cost. The body of a `ObjectMessage` uses Java serialization to serialize it to bytes. The Java serialized form of even small objects is very verbose so takes up a lot of space on the wire, also Java serialization is slow compared to custommarshalling techniques. Only use `ObjectMessage` if you really can't use one of the other message types, i.e. if you really don't know the type of the payload until run-time.

• Avoid `AUTO_ACKNOWLEDGE`. `AUTO_ACKNOWLEDGE` mode requires an acknowledgement to be sent from the server for each message received on the client, this means more traffic on the network. If you can, use `DUPS_OK_ACKNOWLEDGE` or use `CLIENT_ACKNOWLEDGE` or a transacted session and batch up many acknowledgements with one acknowledge/commit.

• Avoid durable messages. By default JMS messages are durable. If you don't really need durable messages then set them to be non-durable. Durable messages incur a lot more overhead in persisting them to storage.

• Batch many sends or acknowledgements in a single transaction. Apache ActiveMQ Artemis will only require a network round trip on the commit, not on every send or acknowledgement.

### 46.3. Other Tunings

There are various other places in Apache ActiveMQ Artemis where we can perform some tuning:

• Use Asynchronous Send Acknowledgements. If you need to send durable messages non transactionally and you need a guarantee that they have reached the server by the time the call to `send()` returns, don't set durable messages to be sent blocking, instead use asynchronous send acknowledgements to get your acknowledgements of send back in a separate stream, see Guarantees of sends and commits for more information on this.

• Use pre-acknowledge mode. With pre-acknowledge mode, messages are acknowledged before they are sent to the client. This reduces the amount of acknowledgement traffic on the wire. For more information on this, see Extra Acknowledge Modes.

• Disable security. You may get a small performance boost by disabling security by setting the `security-enabled` parameter to `false` in `broker.xml`.

• Disable persistence. If you don't need message persistence, turn it off altogether by setting `persistence-enabled` to `false` in `broker.xml`.

• Sync transactions lazily. Setting `journal-sync-transactional` to `false` in `broker.xml` can give you better transactional persistent performance at the expense of some possibility of loss of transactions on failure. See Guarantees of sends and commits for more information.

• Sync non transactional lazily. Setting `journal-sync-non-transactional` to `false` in `broker.xml` can give you better non-transactional persistent performance at the expense of some possibility of loss of durable messages on failure. See Guarantees of sends and commits for more information.

• Send messages non blocking. Setting `block-on-durable-send` and `block-on-non-durable-send` to `false` in the jms config (if you're using JMS and JNDI) or directly on the ServerLocator. This means you don't have to wait a whole network round trip for every message sent. See Guarantees of sends and commits for more information.
• If you have very fast consumers, you can increase consumer-window-size. This effectively disables consumer flow control.

• Use the core API not JMS. Using the JMS API you will have slightly lower performance than using the core API, since all JMS operations need to be translated into core operations before the server can handle them. If using the core API try to use methods that take `SimpleString` as much as possible. `SimpleString`, unlike `java.lang.String` does not require copying before it is written to the wire, so if you re-use `SimpleString` instances between calls then you can avoid some unnecessary copying.

• If using frameworks like Spring, configure destinations permanently broker side and enable `cacheDestinations` on the client side. See the Setting The Destination Cache for more information on this.

### 46.4. Tuning Transport Settings

• TCP buffer sizes. If you have a fast network and fast machines you may get a performance boost by increasing the TCP send and receive buffer sizes. See the Configuring the Transport for more information on this.

    Note that some operating systems like later versions of Linux include TCP auto-tuning and setting TCP buffer sizes manually can prevent auto-tune from working and actually give you worse performance!

• Increase limit on file handles on the server. If you expect a lot of concurrent connections on your servers, or if clients are rapidly opening and closing connections, you should make sure the user running the server has permission to create sufficient file handles.

    This varies from operating system to operating system. On Linux systems you can increase the number of allowable open file handles in the file `/etc/security/limits.conf` e.g. add the lines

    ```
    serveruser   soft  nofile  20000
    serveruser   hard  nofile  20000
    ```

    This would allow up to 20000 file handles to be open by the user `serveruser`.

• Use `batch-delay` and set `direct-deliver` to false for the best throughput for very small messages. Apache ActiveMQ Artemis comes with a preconfigured connector/acceptor pair (netty-throughput) in `broker.xml` and JMS connection factory (ThroughputConnectionFactory) in `activemq-jms.xml` which can be used to give the very best throughput, especially for small messages. See the Configuring the Transport for more information on this.

### 46.5. Tuning the VM

We highly recommend you use the latest Java JVM for the best performance. We test internally using the Sun JVM, so some of these tunings won’t apply to JDKs from other providers (e.g. IBM or JRockit)
• Memory settings. Give as much memory as you can to the server. Apache ActiveMQ Artemis can run in low memory by using paging (described in Paging) but if it can run with all queues in RAM this will improve performance. The amount of memory you require will depend on the size and number of your queues and the size and number of your messages. Use the JVM arguments \(-Xms\) and \(-Xmx\) to set server available RAM. We recommend setting them to the same high value.

When under periods of high load, it is likely that Artemis will be generating and destroying lots of objects. This can result in a build up of stale objects. To reduce the chance of running out of memory and causing a full GC (which may introduce pauses and unintentional behaviour), it is recommended that the max heap size \(-Xmx\) for the JVM is set at least to 5 x the global-max-size of the broker. As an example, in a situation where the broker is under high load and running with a global-max-size of 1GB, it is recommended the max heap size is set to 5GB.

46.6. Avoiding Anti-Patterns

• Re-use connections / sessions / consumers / producers. Probably the most common messaging anti-pattern we see is users who create a new connection/session/producer for every message they send or every message they consume. This is a poor use of resources. These objects take time to create and may involve several network round trips. Always re-use them.

> Spring’s JmsTemplate is known to use this anti-pattern. It can only safely be used with a connection pool (e.g. in a Java EE application server using JCA), and even then it should only be used for sending messages. It cannot be safely be used for synchronously consuming messages, even with a connection pool. If you need a connection pool take a look at this which was forked from the ActiveMQ code-base into its own project with full support for JMS 2.

• Avoid fat messages.Verbose formats such as XML take up a lot of space on the wire and performance will suffer as result. Avoid XML in message bodies if you can.

• Don’t create temporary queues for each request. This common anti-pattern involves the temporary queue request-response pattern. With the temporary queue request-response pattern a message is sent to a target and a reply-to header is set with the address of a local temporary queue. When the recipient receives the message they process it then send back a response to the address specified in the reply-to. A common mistake made with this pattern is to create a new temporary queue on each message sent. This will drastically reduce performance. Instead the temporary queue should be re-used for many requests.

• Don’t use Message-Driven Beans for the sake of it. As soon as you start using MDBs you are greatly increasing the codepath for each message received compared to a straightforward message consumer, since a lot of extra application server code is executed. Ask yourself do you really need MDBs? Can you accomplish the same task using just a normal message consumer?

46.7. Troubleshooting
46.7.1. UDP not working

In certain situations UDP used on discovery may not work. Typical situations are:

1. The nodes are behind a firewall. If your nodes are on different machines then it is possible that the firewall is blocking the multicasts. You can test this by disabling the firewall for each node or adding the appropriate rules.

2. You are using a home network or are behind a gateway. Typically home networks will redirect any UDP traffic to the Internet Service Provider which is then either dropped by the ISP or just lost. To fix this you will need to add a route to the firewall/gateway that will redirect any multicast traffic back on to the local network instead.

3. All the nodes are in one machine. If this is the case then it is a similar problem to point 2 and the same solution should fix it. Alternatively, you could add a multicast route to the loopback interface. On Linux the command would be:

```bash
# you should run this as root
route add -net 224.0.0.0 netmask 240.0.0.0 dev lo
```

This will redirect any traffic directed to the 224.0.0.0 to the loopback interface. This will also work if you have no network at all. On Mac OS X, the command is slightly different:

```bash
sudo route add 224.0.0.0 127.0.0.1 -netmask 240.0.0.0
```
Chapter 47. Performance Tools

Artemis provides some built-in performance test tools based on the JMS 2 API to help users (and developers) to stress test a configured Artemis broker instance in different scenarios.

These command-line tools won't represent a full-fat benchmark (such as Open Messaging), but can be used as building blocks to produce one. They are also quite useful on their own.

In summary, the provided perf tools are:

1. **producer** tool: it can generate both all-out throughput or target-rate load, using BytesMessage of a configured size
2. **consumer** tool: it uses a MessageListener to consume messages sent by the producer command
3. **client** tools: it packs both tools as a single command

Most users will just need the client tool, but the producer and consumer tools allow performing tests in additional scenario(s):

- delaying consumer start, in order to cause the broker to page
- running producers and consumers on different machines
- …

The examples below (running on a 64 bit Linux 5.14 with Intel® Core™ i7-9850H CPU @ 2.60GHz × 12 with Turbo Boost disabled, 32 GB of RAM and SSD) show different use cases of increasing complexity. As they progress, some internal architectural details of the tool and the configuration options supported, are explored.

47.1. Case 1: Single producer Single consumer over a queue

This is the simplest possible case: running a load test with 1 producer and 1 consumer on a non-durable queue TEST_QUEUE, using non-persistent 1024 bytes long (by default) messages, using auto-acknowledge.

Let's see what happens after typing:

```
$ ./artemis perf client queue://TEST_QUEUE
Connection brokerURL = tcp://localhost:61616
2022-01-18 10:30:54,535 WARN [org.apache.activemq.artemis.core.client] AMQ212053:
CompletionListener/SendAcknowledgementHandler used with confirmationWindowSize=-1.
```
Enable confirmationWindowSize to receive acks from server!
---
--- warmup false
--- sent: 7316 msg/sec
--- blocked: 6632 msg/sec
--- completed: 7320 msg/sec
--- received: 7317 msg/sec
# ...

The test keeps on running, until SIGTERM or SIGINT signals are sent to the Java process (on Linux Console it translates into pressing CTRL + C). Before looking what the metrics mean, there’s an initial WARN log that shouldn’t be ignored:

```
WARN [org.apache.activemq.artemis.core.client] AMQ212053:
CompletionListener/SendAcknowledgementHandler used with confirmationWindowSize=-1.
Enable confirmationWindowSize to receive acks from server!
```

It shows two things:

1. the load generator uses async message producers
2. confirmationWindowSize is an Artemis CORE protocol specific setting; the perf commands uses CORE as the default JMS provider

47.1.1. Live Latency Console Reporting

The perf client command can report on Console different latency percentiles metrics by adding --show-latency to the command arguments, but in order to obtain meaningful metrics, we need to address WARN by setting confirmationWindowSize on the producer url, setting --consumer-url to save applying the same configuration for consumer(s).

In short, the command is using these additional parameters:

```
```

Running it

```
--- warmup false
--- sent: 8114 msg/sec
--- blocked: 8114 msg/sec
--- completed: 8114 msg/sec
--- received: 8113 msg/sec
--- send ack time: mean: 113.01 us - 50.00%: 106.00 us - 90.00%: 142.00 us - 99.00%: 204.00 us - 99.90%: 371.00 us - 99.99%: 3455.00 us - max:
```
Some notes:

1. WARN message is now gone
2. send ack time and transfer time statistics are printed at second interval
3. total and aggregated metrics are printed on test completion (more on this later)

The meaning of the live latency statistics are:

**send ack time**
percentiles of latency to acknowledge sent messages

**transfer time**
percentiles of latency to transfer messages from producer(s) to consumer(s)

The **perf** commands uses JMS 2 async message producers that allow the load generator to accumulate in-flight sent messages and depending on the protocol implementation, may block its producer thread due to producer flow control. e.g: the Artemis CORE protocol can block producers threads to refill producers credits, while the QPID-JMS won't.

The perf tool is implementing its own in-flight sent requests tracking and can be configured to limit the amount of pending sent messages, while reporting the rate by which producers are "blocked" awaiting completions

---

**Producer threads are blocked?**

Although the load back-pressure mechanism is non-blocking, given that the load generator cannot push further load while back-pressured by the protocol client, the load is semantically "blocked". This detail is relevant to explain the live rate statistics on Console:

By default, the perf tools (i.e: client and producer) **limits the number of in-flight request to 1**: to
change the default setting users should add `--max-pending` parameter configuration.

Setting `--max-pending 0` will disable the load generator in-flight sent messages limiter, allowing the tool to accumulate an unbounded number of in-flight messages, risking `OutOfMemoryError`.

This is **NOT RECOMMENDED**!

More detail on the metrics:

**warmup**

The generator phase while the statistics sample is collected; warmup duration can be set by setting `--warmup`

**sent**

The message sent rate

**blocked**

The rate of attempts to send a new message, "blocked" awaiting `--max-pending` refill

**completed**

The rate of message send acknowledgements received by producer(s)

**received**

The rate of messages received by consumer(s)

47.1.2. How to read the live statistics?

+ The huge amount of `blocked` vs `sent` means that the broker wasn't fast enough to refill the single `--max-pending` budget before sending a new message. + It can be changed into:

```
--max-pending 100
```

```
# first samples shows very BAD performance because client JVM is still warming up
--- warmup true
--- sent: 27366 msg/sec
--- blocked: 361 msg/sec
--- completed: 27305 msg/sec
--- received: 26195 msg/sec
--- send ack time: mean: 1743.39 us - 50.00%: 1551.00 us - 90.00%: 3119.00 us - 99.00%: 5215.00 us - 99.90%: 8575.00 us - 99.99%: 8703.00 us - max: 23679.00 us
--- transfer time: mean: 11860.32 us - 50.00%: 11583.00 us - 90.00%: 18539.00 us - 99.00%: 24319.00 us - max: 31615.00 us
```
Some notes on the results:

• we now have a reasonable blocked/sent ratio (< ~10%)
• sent rate has improved ten-fold if compared to previous results

And on the SUMMARY statistics:

• total counters include measurements collected with warmup true
• aggregated latencies don’t include measurements collected with warmup true

47.1.3. How to compare latencies across tests?

The Console output format isn’t designed for easy latency comparisons, however the perf commands expose --hdr <hdr file name> parameter to produce a HDR Histogram compatible report that can be opened with different visualizers + eg Online HdrHistogram Log Analyzer, HdrHistogramVisualizer or HistogramLogAnalyzer.

Any latency collected trace on this guide is going to use Online HdrHistogram Log Analyzer as HDR Histogram visualizer tool.

Below is the visualization of the HDR histograms collected while adding to the previous benchmark...
Whole test execution shows tagged latencies, to distinguish warmup ones:

![Latency by Percentile Distribution](image1)

Filtering out warmup latencies, it looks like

![Latency by Percentile Distribution](image2)

Latency results shows that at higher percentiles transfer latency is way higher than the sent one (reminder: sent it’s the time to acknowledge sent messages), probably meaning that some queuing-up is happening on the broker.

In order to test this theory we switch to target rate tests.

47.2. Case 2: Target Rate Single producer Single consumer over a queue

perf client and perf producer tools allow specifying a target rate to schedule producer(s) requests: adding

```
--rate <msg/sec integer value>
```

The previous example last run shows that --max-pending 100 guarantees < 10% blocked/sent messages with aggregated latencies

```
--- aggregated send time:    mean: 1056.09 us - 50.00%: 1003.00 us - 90.00%: 1423.00 us - 99.00%: 1639.00 us - 99.90%: 4287.00 us - 99.99%: 7103.00 us - max: 19583.00 us
--- aggregated transfer time: mean: 18647.51 us - 50.00%: 10751.00 us - 90.00%: 54271.00 us - 99.00%: 84991.00 us - 99.90%: 90111.00 us - 99.99%: 93183.00 us -
```
We would like to lower transfer time sub-millisecond; let's try by running a load test with ~30% of the max perceived sent rate, by setting:

```
--rate 30000 --hdr /tmp/30K.hdr
```

The whole command is then:

```
$ ./artemis perf client --rate 30000 --hdr /tmp/30K.hdr --warmup 20 --max-pending 100

# ... after 20 warmup seconds ...

--- warmup false
--- sent: 30302 msg/sec
--- blocked: 0 msg/sec
--- completed: 30302 msg/sec
--- received: 30303 msg/sec
--- send delay time: mean: 24.20 us - 50.00%: 21.00 us - 90.00%: 54.00 us - 99.00%: 731.00 us
--- send ack time: mean: 150.48 us - 50.00%: 120.00 us - 90.00%: 172.00 us - 99.00%: 3247.00 us
--- transfer time: mean: 171.53 us - 50.00%: 135.00 us - 90.00%: 194.00 us - 99.00%: 3183.00 us

# CTRL + C

--- SUMMARY
--- result: success
--- total sent: 1216053
--- total blocked: 845
--- total completed: 1216053
--- total received: 1216053
--- aggregated delay send time: mean: 35.84 us - 50.00%: 20.00 us - 90.00%: 55.00 us - 99.00%: 116.00 us - 99.90%: 3359.00 us - 99.99%: 5503.00 us - max: 6495.00 us
--- aggregated send time: mean: 147.38 us - 50.00%: 117.00 us - 90.00%: 165.00 us - 99.00%: 991.00 us - 99.90%: 4191.00 us - 99.99%: 5695.00 us - max: 7103.00 us
--- aggregated transfer time: mean: 178.48 us - 50.00%: 134.00 us - 90.00%: 188.00 us - 99.00%: 1359.00 us - 99.90%: 5471.00 us - 99.99%: 8831.00 us - max: 12799.00 us

We've now achieved sub-millisecond transfer latencies until 90.00 percentile. + Opening /tmp/30K.hdr makes easier to see it:
Now send and transfer time looks quite similar and there's no sign of queueing, but...

47.2.1. What delay send time means?

This metric is borrowed from the Coordinated Omission concept, and it measures the delay of producer(s) while trying to send messages at the requested rate.

The source of such delay could be:

- slow responding broker: the load generator reached --max-pending and the expected rate cannot be honored
- client running out of resources (lack of CPU time, GC pauses, etc etc): load generator cannot keep-up with the expected rate because it is just “too fast” for it
- protocol-dependent blocking behaviours: CORE JMS 2 async send can block due to producerWindowSize exhaustion

A sane run of a target rate test should keep delay send time under control or investigation actions must be taken to understand what's the source of the delay. + Let's show it with an example: we've already checked the all-out rate of the broker ie \(\sim 90\text{K msg/sec}\)

By running a --rate 90000 test under the same conditions, latencies will look as

It clearly shows that the load generator is getting delayed and cannot keep-up with the expected rate.

Below is a more complex example involving destinations (auto)generation with "asymmetric" load i.e: the producer number is different from consumer number.
47.3. Case 3: Target Rate load on 10 durable topics, each with 3 producers and 2 unshared consumers

The perf tool can auto generate destinations using

```
--num-destinations <number of destinations to generate>
```

and naming them by using the destination name specified as the seed and an ordered sequence suffix.

eg

```
--num-destinations 3 topic://TOPIC
```

would generate 3 topics: TOPIC0, TOPIC1, TOPIC2.

With the default configuration (without specifying --num-destinations) it would just create TOPIC, without any numerical suffix.

In order to create a load generation on 10 topics, each with 3 producers and 2 unshared consumers:

```
--producers 3 --consumers 2 --num-destinations 10 topic://TOPIC
```

The whole perf client all-out throughput command would be:

```
# same as in the previous cases
./artemis perf client --warmup 20 --max-pending 100 --s
how-latency --url tcp://localhost:61616?confirmationWindowSize=20000 --consumer-url
tcp://localhost:61616
--producers 3 --consumers 2 --num-destinations 10 --durable --persistent
topic://DURABLE_TOPIC
# this last part above is new
```

and it would print...

```
javax.jms.IllegalArgumentException: Cannot create durable subscription - client ID has not been set
```

Given that the generator is creating unshared durable Topic subscriptions, is it mandatory to set a ClientID for each connection used.

The perf client tool creates a connection for each consumer by default and auto-generates both ClientIDs and subscriptions names (as required by the unshared durable Topic subscriptions API). ClientID still requires users to specify Client ID prefixes with --clientID <Client ID prefix> and
takes care to unsubscribe the consumers on test completion.

The complete commands now looks like:

```bash
d--producers 3 --consumers 2 --num-destinations 10 --durable --persistent
--clientID test_id
# after few seconds
---
--- warmup false
--- sent: 74842 msg/sec
--- blocked: 2702 msg/sec
--- completed: 74641 msg/sec
--- received: 146412 msg/sec
--- send ack time: mean: 37366.13 us - 50.00%: 37119.00 us - 90.00%: 46079.00 us - 99.00%: 60895.00 us - 99.90%: 84479.00 us - 99.99%: 94719.00 us - max: 95743.00 us
--- transfer time: mean: 44060.66 us - 50.00%: 43263.00 us - 90.00%: 54527.00 us - 99.00%: 75775.00 us - 99.90%: 87551.00 us - 99.99%: 91135.00 us - max: 91135.00 us
# CTRL + C
--- SUMMARY
--- result: success
--- total sent: 2377653
--- total blocked: 80004
--- total completed: 2377653
--- total received: 4755306
--- aggregated send time: mean: 39423.69 us - 50.00%: 38911.00 us - 90.00%: 49663.00 us - 99.00%: 6047.00 us - 99.90%: 85503.00 us - 99.99%: 101887.00 us - max: 115711.00 us
--- aggregated transfer time: mean: 46216.99 us - 50.00%: 45311.00 us - 90.00%: 57855.00 us - 99.00%: 78335.00 us - 99.90%: 97791.00 us - 99.99%: 113151.00 us - max: 125439.00 us
```

Results shows that transfer time isn’t queuing up, meaning that subscribers are capable to keep-up with the producers: hence a reasonable rate to test could be ~80% of the perceived sent rate ie --rate 60000:

```bash
--producers 3 --consumers 2 --num-destinations 10 --durable --persistent
--clientID test_id
--rate 60000
# after many seconds while running
--- warmup false
--- sent: 55211 msg/sec
--- blocked: 2134 msg/sec
```
What's wrong with the send delay time? Results show that the load generator cannot keep up with the expected rate and it's accumulating a huge delay on the expected scheduled load: let's trying fixing it by adding more producers threads, adding

```bash
--threads <producer threads>
```

By using two producers threads, the command now looks like:

```
--producers 3 --consumers 2 --num-destinations 10 --durable --persistent topic://DURABLE_TOPIC --clientID test_id
--rate 60000 --threads 2
# after few seconds warming up....
--- warmup false
--- sent: 59894 msg/sec
--- blocked: 694 msg/sec
--- completed: 58925 msg/sec
--- received: 114857 msg/sec
--- send delay time: mean: 3189.96 us - 50.00%: 277.00 us - 90.00%: 10623.00 us - 99.00%: 35583.00 us - 99.90%: 47871.00 us - 99.99%: 56063.00 us - max: 58367.00 us
--- send ack time: mean: 31500.93 us - 50.00%: 31231.00 us - 90.00%: 48383.00 us - 99.00%: 65535.00 us - 99.90%: 83455.00 us - 99.99%: 95743.00 us - max: 98303.00 us
--- transfer time: mean: 38151.21 us - 50.00%: 37119.00 us - 90.00%: 104959.00 us - 99.00%: 84479.00 us - 99.90%: 118271.00 us - max: 121855.00 us
```

Send delay time now seems under control, meaning that the load generator need some tuning in order to work at its best.
Chapter 48. Thread management

This chapter describes how Apache ActiveMQ Artemis uses and pools threads and how you can manage them.

First we’ll discuss how threads are managed and used on the server side, then we’ll look at the client side.

48.1. Server-Side Thread Management

Each Apache ActiveMQ Artemis Server maintains a single thread pool for general use, and a scheduled thread pool for scheduled use. A Java scheduled thread pool cannot be configured to use a standard thread pool, otherwise we could use a single thread pool for both scheduled and non scheduled activity.

Apache ActiveMQ Artemis will, by default, cap its thread pool at three times the number of cores (or hyper-threads) as reported by `Runtime.getRuntime().availableProcessors()` for processing incoming packets. To override this value, you can set the number of threads by specifying the parameter `nioRemotingThreads` in the transport configuration. See the configuring transports for more information on this.

There are also a small number of other places where threads are used directly, we’ll discuss each in turn.

48.1.1. Server Scheduled Thread Pool

The server scheduled thread pool is used for most activities on the server side that require running periodically or with delays. It maps internally to a `java.util.concurrent.ScheduledThreadPoolExecutor` instance.

The maximum number of thread used by this pool is configure in `broker.xml` with the `scheduled-thread-pool-max-size` parameter. The default value is 5 threads. A small number of threads is usually sufficient for this pool.

48.1.2. General Purpose Server Thread Pool

This general purpose thread pool is used for most asynchronous actions on the server side. It maps internally to a `java.util.concurrent.ThreadPoolExecutor` instance.

The maximum number of thread used by this pool is configure in `broker.xml` with the `thread-pool-max-size` parameter.

If a value of -1 is used this signifies that the thread pool has no upper bound and new threads will be created on demand if there are not enough threads available to satisfy a request. If activity later subsides then threads are timed-out and closed.

If a value of n where n is a positive integer greater than zero is used this signifies that the thread pool is bounded. If more requests come in and there are no free threads in the pool and the pool is full then requests will block until a thread becomes available. It is recommended that a bounded
thread pool is used with caution since it can lead to dead-lock situations if the upper bound is chosen to be too low.

The default value for thread-pool-max-size is 30.

See the J2SE javadoc for more information on unbounded (cached), and bounded (fixed) thread pools.

### 48.1.3. Expiry Reaper Thread

A single thread is also used on the server side to scan for expired messages in queues. We cannot use either of the thread pools for this since this thread needs to run at its own configurable priority.

For more information on configuring the reaper, please see message expiry.

### 48.1.4. Asynchronous IO

Asynchronous IO has a thread pool for receiving and dispatching events out of the native layer. You will find it on a thread dump with the prefix ActiveMQ-AIO-poller-pool. Apache ActiveMQ Artemis uses one thread per opened file on the journal (there is usually one).

There is also a single thread used to invoke writes on libaio. We do that to avoid context switching on libaio that would cause performance issues. You will find this thread on a thread dump with the prefix ActiveMQ-AIO-writer-pool.

### 48.2. Client-Side Thread Management

On the client side, Apache ActiveMQ Artemis maintains a single, "global" static scheduled thread pool and a single, "global" static general thread pool for use by all clients using the same classloader in that JVM instance.

The static scheduled thread pool has a maximum size of 5 threads by default. This can be changed using the scheduledThreadPoolMaxSize URI parameter.

The general purpose thread pool has an unbounded maximum size. This is changed using the threadPoolMaxSize URL parameter.

If required Apache ActiveMQ Artemis can also be configured so that each ClientSessionFactory instance does not use these "global" static pools but instead maintains its own scheduled and general purpose pool. Any sessions created from that ClientSessionFactory will use those pools instead. This is configured using the useGlobalPools boolean URL parameter.
Chapter 49. Scheduled Messages

Scheduled messages differ from normal messages in that they won't be delivered until a specified time in the future, at the earliest.

To do this, a special property is set on the message before sending it.

49.1. Scheduled Delivery Property

The property name used to identify a scheduled message is "_AMQ_SCHED_DELIVERY" (or the constant Message.HDR_SCHEDULED_DELIVERY_TIME).

The specified value must be a positive long corresponding to the time the message must be delivered (in milliseconds). An example of sending a scheduled message using the JMS API is as follows.

```java
TextMessage message = session.createTextMessage("This is a scheduled message message which will be delivered in 5 sec.");
message.setLongProperty("_AMQ_SCHED_DELIVERY", System.currentTimeMillis() + 5000);
producer.send(message);

// message will not be received immediately but 5 seconds later
TextMessage messageReceived = (TextMessage) consumer.receive();
```

Scheduled messages can also be sent using the core API, by setting the same property on the core message before sending.

49.2. Example

See the Scheduled Message Example which shows how scheduled messages can be used with JMS.
Chapter 50. Last-Value Queues

Last-Value queues are special queues which discard any messages when a newer message with the same value for a well-defined Last-Value property is put in the queue. In other words, a Last-Value queue only retains the last value.

A typical example for Last-Value queue is for stock prices, where you are only interested by the latest value for a particular stock.

Messages sent to an Last-Value queue without the specified property will be delivered as normal and will never be "replaced".

50.1. Configuration

Last Value Key Configuration

Last-Value queues can be statically configured in broker.xml via the `last-value-key`.

```xml
<address name="foo.bar">
    <multicast>
        <queue name="orders1" last-value-key="reuters_code" />
    </multicast>
</address>
```

Specified on creating a queue by using the CORE api specifying the parameter `lastValue` to `true`.

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```java
Queue queue = session.createQueue("my.destination.name?last-value-key=reuters_code");
Topic topic = session.createTopic("my.destination.name?last-value-key=reuters_code");
```

Address wildcards can be used to configure Last-Value queues for a set of addresses (see here).

```xml
<address-setting match="lastValueQueue">
    <default-last-value-key>reuters_code</default-last-value-key>
</address-setting>
```

By default, `default-last-value-key` is `null`.

Legacy Last Value Configuration

Last-Value queues can also just be configured via the `last-value` boolean property, doing so it will default the last-value-key to `_AMQ_LVQ_NAME_`. 
Specified on creating a queue by using the CORE api specifying the parameter lastValue to true.

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```java
Queue queue = session.createQueue("my.destination.name?last-value=true");
Topic topic = session.createTopic("my.destination.name?last-value=true");
```

Also the default for all queues under and address can be defaulted using the address-setting configuration:

```xml
<address-setting match="lastValueQueue">
  <default-last-value-queue>true</default-last-value-queue>
</address-setting>
```

By default, default-last-value-queue is false.

Note that address-setting last-value-queue config is deprecated, please use default-last-value-queue instead.

### 50.2. Last-Value Property

The property name used to identify the last value is configurable at the queue level mentioned above.

If using the legacy setting to configure an LVQ then the default property ".AMQ_LVQ_NAME" is used (or the constant Message.HDR_LAST_VALUE_NAME from the Core API).

For example, using the sample configuration

```xml
<address name="foo.bar">
  <multicast>
    <queue name="orders1" last-value-key="reuters_code" />
  </multicast>
</address>
```

if two messages with the same value for the Last-Value property are sent to a Last-Value queue, only the latest message will be kept in the queue:
50.3. Forcing all consumers to be non-destructive

It's common to combine last-value queues with non-destructive semantics.

50.4. Clustering

The fundamental ideas behind last-value queues and clustering are at odds with each other.

Clustering was designed as a way to increase message throughput through horizontal scaling. The messages in a clustered queue can be spread across all nodes in the cluster. This allows clients to be distributed across the cluster to leverage the computing resources all the nodes rather than being bottlenecked on a single node.

However, if you wanted to use a last-value queue in a cluster then in order to enforce last-value semantics all messages would be required to go to a queue on a single node. This would effectively nullify the benefits of clustering. Also, the arrival of messages on and and redistribution of those messages from nodes other than the node where the last-value semantics would be enforced would almost certainly impact which message is considered "last."

For these reasons last-value queues are not supported in a traditional cluster. However, it would be possible to use a connection router in front of a cluster (or even a set of non-clustered brokers) to ensure all clients which need to use the same last-value queue are directed to the same node. See the connection router for more details on configuration, etc.

50.5. Example

See the last-value queue example which shows how last value queues are configured and used with JMS.
Chapter 51. Non-Destructive Queues

When a consumer attaches to a queue, the normal behaviour is that messages are sent to that consumer are acquired exclusively by that consumer, and when the consumer acknowledges them, the messages are removed from the queue.

Another common pattern is to have queue "browsers" which send all messages to the browser, but do not prevent other consumers from receiving the messages, and do not remove them from the queue when the browser is done with them. Such a browser is an instance of a "non-destructive" consumer.

If every consumer on a queue is non destructive then we can obtain some interesting behaviours. In the case of a last value queue then the queue will always contain the most up to date value for every key.

A queue can be created to enforce all consumers are non-destructive using the following queue configuration:

```xml
<address name="foo.bar">
    <multicast>
        <queue name="orders1" non-destructive="true" />
    </multicast>
</address>
```

Or on auto-create when using the JMS client by using address parameters when creating the destination used by the consumer.

```java
Queue queue = session.createQueue("my.destination.name?non-destructive=true");
Topic topic = session.createTopic("my.destination.name?non-destructive=true");
```

Also the default for all queues under and address can be defaulted using the address-setting configuration:

```xml
<address-setting match="nonDestructiveQueue">
    <default-non-destructive>true</default-non-destructive>
</address-setting>
```

By default, default-non-destructive is false.

51.1. Limiting the Size of the Queue

For queues other than last-value queues, having only non-destructive consumers could mean that messages would never get deleted, leaving the queue to grow without constraint. To prevent this you can use the ability to set a default expiry-delay. See expiry-delay for more details on this. You could also use a ring queue.
Chapter 52. Ring Queue

Queues operate with first-in, first-out (FIFO) semantics which means that messages, in general, are added to the "tail" of the queue and removed from the "head." A "ring" queue is a special type of queue with a fixed size. The fixed size is maintained by removing the message at the head of the queue when the number of messages on the queue reaches the configured size.

For example, consider a queue configured with a ring size of 3 and a producer which sends the messages A, B, C, & D in that order. Once C is sent the number of messages in the queue will be 3 which is the same as the configured ring size. We can visualize the queue growth like this...

After A is sent:

head/tail -> |---|

After B is sent:

head -> |---|

tail -> |---|

After C is sent:

head -> |---|

tail -> |---|

When D is sent it will be added to the tail of the queue and the message at the head of the queue (i.e. A) will be removed so the queue will look like this:

head -> |---|

tail -> |---|
This example covers the most basic use case with messages being added to the tail of the queue. However, there are a few other important use cases involving:

- Messages in delivery & rollbacks
- Scheduled messages
- Paging

However, before we get to those use cases let’s look at the basic configuration of a ring queue.

### 52.1. Configuration

There are 2 parameters related to ring queue configuration.

The `ring-size` parameter can be set directly on the `queue` element. The default value comes from the `default-ring-size address-setting` (see below).

```
<addresses>
    <address name="myRing">
        <anycast>
            <queue name="myRing" ring-size="3" />
        </anycast>
    </address>
</addresses>
```

The `default-ring-size` is an `address-setting` which applies to queues on matching addresses which don’t have an explicit `ring-size` set. This is especially useful for auto-created queues. The default value is `-1` (i.e. no limit).

```
<address-settings>
    <address-setting match="ring.#">
        <default-ring-size>3</default-ring-size>
    </address-setting>
</address-settings>
```

The `ring-size` may be updated at runtime. If the new `ring-size` is set lower than the previous `ring-size` the broker will not immediately delete enough messages from the head of the queue to enforce the new size. New messages sent to the queue will force the deletion of old messages (i.e. the queue won’t grow any larger), but the queue will not reach its new size until it does so *naturally* through the normal consumption of messages by clients.

### 52.2. Messages in Delivery & Rollbacks

When messages are “in delivery” they are in an in-between state where they are not technically on the queue but they are also not yet acknowledged. The broker is at the consumer’s mercy to either acknowledge such messages or not. In the context of a ring queue, messages which are in-delivery cannot be removed from the queue.
This presents a few dilemmas.

Due to the nature of messages in delivery a client can actually send more messages to a ring queue than it would otherwise permit. This can make it appear that the ring-size is not being enforced properly. Consider this simple scenario:

- Queue foo with ring-size="3"
- 1 Consumer on queue foo
- Message A sent to foo & dispatched to consumer
  - messageCount=1, deliveringCount=1
- Message B sent to foo & dispatched to consumer
  - messageCount=2, deliveringCount=2
- Message C sent to foo & dispatched to consumer
  - messageCount=3, deliveringCount=3
- Message D sent to foo & dispatched to consumer
  - messageCount=4, deliveringCount=4

The messageCount for foo is now 4, one greater than the ring-size of 3! However, the broker has no choice but to allow this because it cannot remove messages from the queue which are in delivery.

Now consider that the consumer is closed without actually acknowledging any of these 4 messages. These 4 in-delivery, unacknowledged messages will be cancelled back to the broker and added to the head of the queue in the reverse order from which they were consumed. This, of course, will put the queue over its configured ring-size. Therefore, since a ring queue prefers messages at the tail of the queue over messages at the head it will keep B, C, & D and delete A (since A was the last message added to the head of the queue).

Transaction or core session rollbacks are treated the same way.

If you wish to avoid these kinds of situations and you're using the core client directly or the core JMS client you can minimize messages in delivery by reducing the size of consumerWindowSize (1024 * 1024 bytes by default).

### 52.3. Scheduled Messages

When a scheduled message is sent to a queue it isn't immediately added to the tail of the queue like normal messages. It is held in an intermediate buffer and scheduled for delivery onto the head of the queue according to the details of the message. However, scheduled messages are nevertheless reflected in the message count of the queue. As with messages which are in delivery this can make it appear that the ring queue's size is not being enforced. Consider this simple scenario:

- Queue foo with ring-size="3"
- At 12:00 message A sent to foo scheduled for 12:05
  - messageCount=1, scheduledCount=1
At 12:01 message B sent to foo
messageCount=2, scheduledCount=1
At 12:02 message C sent to foo
messageCount=3, scheduledCount=1
At 12:03 message D sent to foo
messageCount=4, scheduledCount=1

The messageCount for foo is now 4, one greater than the ring-size of 3! However, the scheduled message is not technically on the queue yet (i.e. it is on the broker and scheduled to be put on the queue). When the scheduled delivery time for 12:05 comes the message will put on the head of the queue, but since the ring queue’s size has already been reach the scheduled message A will be removed.

52.4. Paging

Similar to scheduled messages and messages in delivery, paged messages don’t count against a ring queue’s size because messages are actually paged at the address level, not the queue level. A paged message is not technically on a queue although it is reflected in a queue’s messageCount.

It is recommended that paging is not used for addresses with ring queues. In other words, ensure that the entire address will be able to fit into memory or use the DROP, BLOCK or FAIL address-full-policy.
Chapter 53. Retroactive Addresses

A "retroactive" address is an address that will preserve messages sent to it for queues which will be created on it in the future. This can be useful in, for example, publish-subscribe use cases where clients want to receive the messages sent to the address before they actually connected and created their multicast "subscription" queue. Typically messages sent to an address before a queue was created on it would simply be unavailable to those queues, but with a retroactive address a fixed number of messages can be preserved by the broker and automatically copied into queues subsequently created on the address. This works for both anycast and multicast queues.

53.1. Internal Retroactive Resources

To implement this functionality the broker will create 4 internal resources for each retroactive address:

1. A non-exclusive divert to grab the messages from the retroactive address.
2. An address to receive the messages from the divert.
3. Two ring queues to hold the messages sent to the address by the divert - one for anycast and one for multicast. The general caveats for ring queues still apply here. See the chapter on ring queues for more details.

These resources are important to be aware of as they will show up in the web console and other management or metric views. They will be named according to the following pattern:

```
<internal-naming-prefix><delimiter><source-address><delimiter>(divert|address|queue<delimiter>(anycast|multicast))<delimiter>retr
```

For example, if an address named `myAddress` had a retroactive-message-count of 10 and the default internal-naming-prefix (i.e. `$.artemis.internal`) and the default delimiter (i.e. `.`) were being used then resources with these names would be created:

1. A divert on `myAddress` named `$.artemis.internal.myAddress.divert.retro`
2. An address named `$.artemis.internal.myAddress.address.retro`
3. A multicast queue on the address from step #2 named `$.artemis.internal.myAddress.queue.multicast.retro` with a ring-size of 10.
4. An anycast queue on the address from step #2 named `$.artemis.internal.myAddress.queue.anycast.retro` with a ring-size of 10.

This pattern is important to note as it allows one to configure address-settings if necessary. To configure custom address-settings you'd use a match like:

```
.*.*.<source-address).*.*.retro
```
Using the same example as above the match would be:

```
*./*.myAddress.*.retro
```

Changing the broker's internal-naming-prefix once these retroactive resources are created will break the retroactive functionality.

### 53.2. Configuration

To configure an address to be "retroactive" simply configure the `retroactive-message-count` address-setting to reflect the number of messages you want the broker to preserve, e.g.:

```
<address-settings>
  <address-setting match="orders">
    <retroactive-message-count>100</retroactive-message-count>
  </address-setting>
</address-settings>
```

The value for `retroactive-message-count` can be updated at runtime either via `broker.xml` or via the management API just like any other address-setting. However, if you reduce the value of `retroactive-message-count` an additional administrative step will be required since this functionality is implemented via ring queues. This is because a ring queue whose ring-size is reduced will not automatically delete messages from the queue to meet the new ring-size in order to avoid unintended message loss. Therefore, administrative action will be required in this case to manually reduce the number of messages in the ring queue via the management API.
Chapter 54. Exclusive Queues

Exclusive queues are special queues which dispatch all messages to only one consumer at a time.

This is useful when you want all messages to be processed serially but you can't or don't want to use Message Grouping.

An example might be orders sent to an address and you need to consume them in the exact same order they were produced.

Obviously exclusive queues have a draw back that you cannot scale out the consumers to improve consumption as only one consumer would technically be active. Here we advise that you look at message groups first.

54.1. Configuring Exclusive Queues

Exclusive queues can be statically configured using the exclusive boolean property:

```
<address name="foo.bar">
    <multicast>
        <queue name="orders1" exclusive="true"/>
    </multicast>
</address>
```

Specified on creating a Queue by using the CORE api specifying the parameter exclusive to true.

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```
Queue queue = session.createQueue("my.destination.name?exclusive=true");
Topic topic = session.createTopic("my.destination.name?exclusive=true");
```

Also the default for all queues under and address can be defaulted using the address-setting configuration:

```
<address-setting match="lastValueQueue">
    <default-exclusive-queue>true</default-exclusive-queue>
</address-setting>
```

By default, default-exclusive-queue is false. Address wildcards can be used to configure exclusive queues for a set of addresses.

54.2. Example

See the exclusive queue example which shows how exclusive queues are configured and used with
Chapter 55. Message Grouping

Message groups are sets of messages that have the following characteristics:

- Messages in a message group share the same group id, i.e. they have same group identifier property (JMSXGroupID for JMS, _AMQ_GROUP_ID for Apache ActiveMQ Artemis Core API).
- Messages in a message group are always consumed by the same consumer, even if there are many consumers on a queue. They pin all messages with the same group id to the same consumer. If that consumer closes another consumer is chosen and will receive all messages with the same group id.

Message groups are useful when you want all messages for a certain value of the property to be processed serially by the same consumer.

An example might be orders for a certain stock. You may want orders for any particular stock to be processed serially by the same consumer. To do this you can create a pool of consumers (perhaps one for each stock, but less will work too), then set the stock name as the value of the _AMQ_GROUP_ID property.

This will ensure that all messages for a particular stock will always be processed by the same consumer.

Grouped messages can impact the concurrent processing of non-grouped messages due to the underlying FIFO semantics of a queue. For example, if there is a chunk of 100 grouped messages at the head of a queue followed by 1,000 non-grouped messages then all the grouped messages will need to be sent to the appropriate client (which is consuming those grouped messages serially) before any of the non-grouped messages can be consumed. The functional impact in this scenario is a temporary suspension of concurrent message processing while all the grouped messages are processed. This can be a performance bottleneck so keep it in mind when determining the size of your message groups, and consider whether or not you should isolate your grouped messages from your non-grouped messages.

55.1. Using Core API

The property name used to identify the message group is "_AMQ_GROUP_ID" (or the constant MessageImpl.HDR_GROUP_ID). Alternatively, you can set autogroup to true on the SessionFactory which will pick a random unique id.

55.2. Using JMS

The property name used to identify the message group is JMSXGroupID.

```java
// send 2 messages in the same group to ensure the same
// consumer will receive both
Message message = ...
message.setStringProperty("JMSXGroupID", "Group-0");
```
Alternatively, you can set autogroup to true on the ActiveMQConnectionFactory which will pick a random unique id. This can also be set in the JNDI context environment, e.g. jndi.properties. Here's a simple example using the “ConnectionFactory” connection factory which is available in the context by default:

```java
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?autoGroup=true
```

Alternatively you can set the group id via the connection factory. All messages sent with producers created via this connection factory will set the JMSXGroupID to the specified value on all messages sent. This can also be set in the JNDI context environment, e.g. jndi.properties. Here's a simple example using the "ConnectionFactory" connection factory which is available in the context by default:

```java
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=tcp://localhost:61616?groupID=Group-0
```

### 55.3. Closing a Message Group

You generally don’t need to close a message group, you just keep using it. However, if you really do want to close a group you can add a negative sequence number.

Example:

```java
Message message = session.createTextMessage("<foo>hey</foo>");
message.setStringProperty("JMSXGroupID", "Group-0");
message.setIntProperty("JMSXGroupSeq", -1);
...
producer.send(message);
```

This closes the message group so if another message is sent in the future with the same message group ID it will be reassigned to a new consumer.

### 55.4. Notifying Consumer of Group Ownership change

ActiveMQ supports putting a boolean header on the first message sent to a consumer for a particular message group.
To enable this you must set a header key that the broker will use to set the flag.

In the examples we use `JMSXGroupFirstForConsumer` but it can be any header key value you want.

By setting `group-first-key` to `JMSXGroupFirstForConsumer` at the queue level, every time a new group is assigned a consumer the header `JMSXGroupFirstForConsumer` will be set to `true` on the first message.

```xml
<address name="foo.bar">
  <multicast>
    <queue name="orders1" group-first-key="JMSXGroupFirstForConsumer"/>
  </multicast>
</address>
```

Or on auto-create when using the core JMS client by using address parameters when creating the destination used by the consumer.

```java
Queue queue = session.createQueue("my.destination.name?group-first-key=JMSXGroupFirstForConsumer");
Topic topic = session.createTopic("my.destination.name?group-first-key=JMSXGroupFirstForConsumer");
```

Also, the default for all queues under and address can be defaulted using the `address-setting` configuration:

```xml
<address-setting match="my.address">
  <default-group-first-key>JMSXGroupFirstForConsumer</default-group-first-key>
</address-setting>
```

By default, this is off.

### 55.5. Rebalancing Message Groups

Sometimes after new consumers are added you can find that they have no groups assigned, and thus are not being utilised. This is because all the groups are already assigned to existing consumers. However, it is possible to rebalance the groups so that all the consumers are the queue are assigned one or more groups.

At the exact moment of a reset a message to the originally associated consumer could be in flight at the same time a new message for the same group is dispatched to the new associated consumer.

#### 55.5.1. Manually

Use the management API (e.g via the web console) by invoking `resetAllGroups` on the associated queue.
55.5.2. Automatically

By setting `group-rebalance` to `true` at the queue level every time a consumer is added it will trigger a rebalance/reset of the groups.

As noted above, when group rebalance is done there is a risk you may have inflight messages being processed. By default, the broker will continue to dispatch whilst rebalance is occurring. To ensure that inflight messages are processed before dispatch of new messages post rebalance, to different consumers, you can set `group-rebalance-pause-dispatch` to `true` which will cause the dispatch to pause whilst rebalance occurs until all inflight messages are processed.

```xml
<address name="foo.bar">
  <multicast>
    <queue name="orders1" group-rebalance="true" group-rebalance-pause-dispatch="true"/>
  </multicast>
</address>
```

Or on auto-create when using the core JMS client by using address parameters when creating the destination used by the consumer.

```java
Queue queue = session.createQueue("my.destination.name?group-rebalance=true&group-rebalance-pause-dispatch=true");
Topic topic = session.createTopic("my.destination.name?group-rebalance=true&group-rebalance-pause-dispatch=true");
```

Also, the default for all queues under and address can be defaulted using the `address-setting` configuration:

```xml
<address-setting match="my.address">
  <default-group-rebalance>true</default-group-rebalance>
  <default-group-rebalance-pause-dispatch>true</default-group-rebalance-pause-dispatch>
</address-setting>
```

By default, `default-group-rebalance` is `false` meaning this is disabled/off. By default, `default-group-rebalance-pause-dispatch` is `false` meaning this is disabled/off.

55.6. Group Buckets

For handling groups in a queue with bounded memory allowing better scaling of groups, you can enable group buckets, essentially the group id is hashed into a bucket instead of keeping track of every single group id.

Setting `group-buckets` to `-1` keeps default behaviour which means the queue keeps track of every group but suffers from unbounded memory use.
Setting **group-buckets** to 0 disables grouping (0 buckets), on a queue. This can be useful on a multicast address, where many queues exist but one queue you may not care for ordering and prefer to keep round robin behaviour.

There is a number of ways to set **group-buckets**.

```xml
<address name="foo.bar">
  <multicast>
    <queue name="orders1" group-buckets="1024"/>
  </multicast>
</address>
```

Specified on creating a Queue by using the CORE api specifying the parameter **group-buckets** to 20.

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```java
Queue queue = session.createQueue("my.destination.name?group-buckets=1024");
Topic topic = session.createTopic("my.destination.name?group-buckets=1024");
```

Also the default for all queues under and address can be defaulted using the **address-setting** configuration:

```xml
<address-setting match="my.bucket.address">
  <default-group-buckets>1024</default-group-buckets>
</address-setting>
```

By default, **default-group-buckets** is -1 this is to keep compatibility with existing default behaviour.

Address **wildcards** can be used to configure group-buckets for a set of addresses.

**55.7. Example**

See the [Message Group Example](#) which shows how message groups are configured and used with JMS and via a connection factory.

**55.8. Clustered Grouping**

Before looking at the details for configuring clustered grouping support it is worth examining the idea of clustered grouping as a whole. In general, combining clustering and message grouping is a poor choice because the fundamental ideas of grouped (i.e. ordered) messages and horizontal scaling through clustering are essentially at odds with each other.

Message grouping enforces ordered message consumption. Ordered message consumption requires that each message be fully consumed and acknowledged before the next message in the group is consumed. This results in *serial* message processing (i.e. no concurrency).
However, the idea of clustering is to scale brokers horizontally in order to increase message throughput by adding consumers which can process messages concurrently. But since the message groups are ordered the messages in each group cannot be consumed concurrently which defeats the purpose of horizontal scaling.

**Clustered grouping is not recommended** for these reasons.

However, if you’ve evaluated your overall use-case with these design caveats in mind and have determined that clustered grouping is still viable then read on for all the configuration details and best practices.

### 55.8.1. Clustered Grouping Configuration

Using message groups in a cluster is a bit more complex. This is because messages with a particular group id can arrive on any node so each node needs to know about which group id's are bound to which consumer on which node. The consumer handling messages for a particular group id may be on a different node of the cluster, so each node needs to know this information so it can route the message correctly to the node which has that consumer.

To solve this there is the notion of a grouping handler. Each node will have its own grouping handler and when a messages is sent with a group id assigned, the handlers will decide between them which route the message should take.

Here is a sample config for each type of handler. This should be configured in `broker.xml`.

```xml
<grouping-handler name="my-grouping-handler">
  <type>LOCAL</type>
  <address>jms</address>
  <timeout>5000</timeout>
</grouping-handler>

<grouping-handler name="my-grouping-handler">
  <type>REMOTE</type>
  <address>jms</address>
  <timeout>5000</timeout>
</grouping-handler>
```

**type**

Two types of handlers are supported - LOCAL and REMOTE. Each cluster should choose 1 node to have a LOCAL grouping handler and all the other nodes should have REMOTE handlers. It’s the LOCAL handler that actually makes the decision as to what route should be used, all the other REMOTE handlers converse with this.

**address**

Refers to a cluster connection and the address it uses. Refer to the clustering section on how to configure clusters.
timeout

How long to wait for a decision to be made. An exception will be thrown during the send if this timeout is reached, this ensures that strict ordering is kept.

The decision as to where a message should be routed to is initially proposed by the node that receives the message. The node will pick a suitable route as per the normal clustered routing conditions, i.e. round robin available queues, use a local queue first and choose a queue that has a consumer. If the proposal is accepted by the grouping handlers the node will route messages to this queue from that point on, if rejected an alternative route will be offered and the node will again route to that queue indefinitely. All other nodes will also route to the queue chosen at proposal time. Once the message arrives at the queue then normal single server message group semantics take over and the message is pinned to a consumer on that queue.

You may have noticed that there is a single point of failure with the single local handler. If this node crashes then no decisions will be able to be made. Any messages sent will be not be delivered and an exception thrown. To avoid this happening Local Handlers can be replicated on another backup node. Simple create your back up node and configure it with the same Local handler.

55.8.2. Clustered Grouping Best Practices

Some best practices should be followed when using clustered grouping:

1. Make sure your consumers are distributed evenly across the different nodes if possible. This is only an issue if you are creating and closing consumers regularly. Since messages are always routed to the same queue once pinned, removing a consumer from this queue may leave it with no consumers meaning the queue will just keep receiving the messages. Avoid closing consumers or make sure that you always have plenty of consumers, i.e., if you have 3 nodes have 3 consumers.

2. Use durable queues if possible. If queues are removed once a group is bound to it, then it is possible that other nodes may still try to route messages to it. This can be avoided by making sure that the queue is deleted by the session that is sending the messages. This means that when the next message is sent it is sent to the node where the queue was deleted meaning a new proposal can successfully take place. Alternatively you could just start using a different group id.

3. Always make sure that the node that has the Local Grouping Handler is replicated. These means that on failover grouping will still occur.

4. In case you are using group-timeouts, the remote node should have a smaller group-timeout with at least half of the value on the main coordinator. This is because this will determine how often the last-time-use value should be updated with a round trip for a request to the group between the nodes.

55.8.3. Clustered Grouping Example

See the Clustered Grouping Example which shows how to configure message groups with a ActiveMQ Artemis Cluster.
Chapter 56. Consumer Priority

Consumer priorities allow you to ensure that high priority consumers receive messages while they are active.

Normally, active consumers connected to a queue receive messages from it in a round-robin fashion. When consumer priorities are in use, messages are delivered round-robin if multiple active consumers exist with the same high priority.

Messages will only going to lower priority consumers when the high priority consumers do not have credit available to consume the message, or those high priority consumers have declined to accept the message (for instance because it does not meet the criteria of any selectors associated with the consumer).

Where a consumer does not set, the default priority 0 is used.

56.1. Core

56.1.1. JMS Example

When using the JMS Client you can set the priority to be used, by using address parameters when creating the destination used by the consumer.

```java
Queue queue = session.createQueue("my.destination.name?consumer-priority=50");
Topic topic = session.createTopic("my.destination.name?consumer-priority=50");
consumer = session.createConsumer(queue);
```

The range of priority values is \(-2^{31}\) to \(2^{31}-1\).

56.2. OpenWire

56.2.1. JMS Example

The priority for a consumer is set using Destination Options as follows:

```java
queue = new ActiveMQQueue("TEST.QUEUE?consumer.priority=10");
consumer = session.createConsumer(queue);
```

Because of the limitation of OpenWire, the range of priority values is: 0 to 127. The highest priority is 127.

56.3. AMQP

In AMQP 1.0 the priority of the consumer is set in the properties map of the attach frame where the
broker side of the link represents the sending side of the link.

The key for the entry must be the literal string priority, and the value of the entry must be an integral number in the range $-2^{31}$ to $2^{31} - 1$. 
Chapter 57. Message Expiry

Messages can be set with an optional *time to live* when sending them.

Apache ActiveMQ Artemis will not deliver a message to a consumer after it's time-to-live has been exceeded. If the message hasn't been delivered by the time that time-to-live is reached the server can discard it.

Apache ActiveMQ Artemis's addresses can be assigned an expiry address so that, when messages are expired, they are removed from the queue and sent to the expiry address. Many different queues can be bound to an expiry address. These *expired* messages can later be consumed for further inspection.

57.1. Core API

Using Apache ActiveMQ Artemis Core API, you can set an expiration time directly on the message:

```java
// message will expire in 5000ms from now
message.setExpiration(System.currentTimeMillis() + 5000);
```

JMS MessageProducer allows to set a TimeToLive for the messages it sent:

```java
// messages sent by this producer will be retained for 5s (5000ms) before expiration
producer.setTimeToLive(5000);
```

Expired messages get special properties plus this additional property:

`_AMQ_ACTUAL_EXPIRY`

- a Long property containing the *actual expiration time* of the expired message

57.2. Configuring Expiry Delay

Default expiry delay can be configured in the address-setting configuration:

```xml
<address-setting match="exampleQueue">
  <expiry-address>expiryQueue</expiry-address>
  <expiry-delay>10</expiry-delay>
</address-setting>
```

`expiry-delay` defines the expiration time in milliseconds that will be used for messages which are using the default expiration time (i.e. 0).

For example, if `expiry-delay` is set to "10" and a message which is using the default expiration time...
(i.e. 10) arrives then its expiration time of "0" will be changed to "10." However, if a message which is using an expiration time of "20" arrives then its expiration time will remain unchanged. Setting `expiry-delay` to "-1" will disable this feature.

The default is -1.

If `expiry-delay` is not set then minimum and maximum expiry delay values can be configured in the address-setting configuration.

```xml
<address-setting match="exampleQueue">
  <min-expiry-delay>10</min-expiry-delay>
  <max-expiry-delay>100</max-expiry-delay>
</address-setting>
```

Semantics are as follows:

- Messages without an expiration will be set to `max-expiry-delay`. If `max-expiry-delay` is not defined then the message will be set to `min-expiry-delay`. If `min-expiry-delay` is not defined then the message will not be changed.
- Messages with an expiration above `max-expiry-delay` will be set to `max-expiry-delay`
- Messages with an expiration below `min-expiry-delay` will be set to `min-expiry-delay`
- Messages with an expiration within `min-expiry-delay` and `max-expiry-delay` range will not be changed.
- Any value set for `expiry-delay` other than the default (i.e. -1) will override the aforementioned min/max settings.

The default for both `min-expiry-delay` and `max-expiry-delay` is -1 (i.e. disabled).

57.3. Configuring Expiry Addresses

Expiry address are defined in the address-setting configuration:

```xml
<!-- expired messages in exampleQueue will be sent to the expiry address expiryQueue -->
<address-setting match="exampleQueue">
  <expiry-address>expiryQueue</expiry-address>
</address-setting>
```

If messages are expired and no expiry address is specified, messages are simply removed from the queue and dropped. Address wildcards can be used to configure expiry address for a set of addresses.

If a wildcard is used to configure the expiry address for a set of addresses and you want to unset the expiry address for a particular address (or set of addresses) then you can do so, e.g.:
57.4. Configuring Automatic Creation of Expiry Resources

It's common to segregate expired messages by their original address. For example, a message sent to the stocks address that expired for some reason might be ultimately routed to the EXP.stocks queue, and likewise a message sent to the orders address that expired might be routed to the EXP.orders queue.

Using this pattern can make it easy to track and administrate expired messages. However, it can pose a challenge in environments which predominantly use auto-created addresses and queues. Typically administrators in those environments don't want to manually create an address-setting to configure the expiry-address much less the actual address and queue to hold the expired messages.

The solution to this problem is to set the auto-create-expiry-resources address-setting to true (it's false by default) so that the broker will create the address and queue to deal with the expired messages automatically. The address created will be the one defined by the expiry-address. A MULTICAST queue will be created on that address. It will be named by the address to which the message was previously sent, and it will have a filter defined using the property _AMQ_ORIG_ADDRESS so that it will only receive messages sent to the relevant address. The queue name can be configured with a prefix and suffix. See the relevant settings in the table below:

<table>
<thead>
<tr>
<th>address-setting</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>expiry-queue-prefix</td>
<td>EXP.</td>
</tr>
<tr>
<td>expiry-queue-suffix</td>
<td>(empty string)</td>
</tr>
</tbody>
</table>

Here is an example configuration:

```xml
<address-setting match="#">
  <expiry-address>expiryAddress</expiry-address>
  <auto-create-expiry-resources>true</auto-create-expiry-resources>
  <expiry-queue-prefix></expiry-queue-prefix> <!-- override the default -->
  <expiry-queue-suffix>.EXP</expiry-queue-suffix>
</address-setting>
```

The queue holding the expired messages can be accessed directly either by using the queue's name by itself (e.g. when using the core client) or by using the fully qualified queue name (e.g. when
using a JMS client) just like any other queue. Also, note that the queue is auto-created which means it will be auto-deleted as per the relevant address-settings.

57.5. Configuring The Expiry Reaper Thread

A reaper thread will periodically inspect the queues to check if messages have expired.

The reaper thread can be configured with the following properties in broker.xml

message-expiry-scan-period

How often the queues will be scanned to detect expired messages (in milliseconds, default is 30000ms, set to -1 to disable the reaper thread)

57.6. Example

See the Message Expiration Example which shows how message expiry is configured and used with JMS.
Chapter 58. Large Messages

Apache ActiveMQ Artemis can be configured to store messages as files when these messages are beyond a configured value.

Instead of keeping these messages in memory ActiveMQ Artemis will hold just a thin object on the queues with a reference to a file into a specific folder configured as large-messages-directory.

This is supported on Core Protocol and on the AMQP Protocol.

58.1. Configuring the server

Large messages are stored on a disk directory on the server side, as configured on the main configuration file.

The configuration property `large-messages-directory` specifies where large messages are stored. For JDBC persistence the `large-message-table` should be configured.

```
<configuration xmlns="urn:activemq"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:activemq /schema/artemis-server.xsd">
     ...
     <large-messages-directory>/data/large-messages</large-messages-directory>
     ...
   </core>
</configuration>
```

By default the large message directory is `data/largemessages` and `large-message-table` is configured as "LARGE_MESSAGE_TABLE".

For the best performance we recommend using file store with large messages directory stored on a different physical volume to the message journal or paging directory.

58.2. Configuring the Core Client

Any message larger than a certain size is considered a large message. Large messages will be split up and sent in fragments. This is determined by the URL parameter `minLargeMessageSize`

Apache ActiveMQ Artemis messages are encoded using 2 bytes per character so if the message data is filled with ASCII characters (which are 1 byte) the size of the resulting Apache ActiveMQ Artemis message would roughly double. This is important when calculating the size of a "large" message as it may appear to be less than the `minLargeMessageSize` before it is sent, but it then turns into a "large" message once it is encoded.
The default value is 100KiB.

Configuring the transport directly from the client side will provide more information on how to instantiate the core session factory or JMS connection factory.

## 58.3. Compressed Large Messages on Core Protocol

You can choose to send large messages in compressed form using `compressLargeMessage` URL parameter.

If you specify the boolean URL parameter `compressLargeMessage` as true, the system will use the ZIP algorithm to compress the message body as the message is transferred to the server’s side. Notice that there’s no special treatment at the server’s side, all the compressing and uncompressing is done at the client.

This behavior can be tuned further by setting an optional parameter: `compressionLevel`. This will decide how much the message body should be compressed. `compressionLevel` accepts an integer of -1 or a value between 0-9. The default value is -1 which corresponds to around level 6-7.

If the compressed size of a large message is below `minLargeMessageSize`, it is sent to server as regular messages. This means that the message won’t be written into the server’s large-message data directory, thus reducing the disk I/O.

A higher `compressionLevel` means the message body will get further compressed, but this is at the cost of speed and computational overhead. Make sure to tune this value according to its specific use-case.

## 58.4. Streaming large messages from Core Protocol

Apache ActiveMQ Artemis supports setting the body of messages using input and output streams (java.lang.io)

These streams are then used directly for sending (input streams) and receiving (output streams) messages.

When receiving messages there are 2 ways to deal with the output stream; you may choose to block while the output stream is recovered using the method `ClientMessage.saveOutputStream` or alternatively using the method `ClientMessage.setOutputstream` which will asynchronously write the message to the stream. If you choose the latter the consumer must be kept alive until the message has been fully received.

You can use any kind of stream you like. The most common use case is to send files stored in your disk, but you could also send things like JDBC Blobs, `SocketInputStream`, things you recovered from HTTPRequests etc. Anything as long as it implements `java.io.InputStream` for sending messages or `java.io.OutputStream` for receiving them.
58.4.1. Streaming over Core API

The following table shows a list of methods available at `ClientMessage` which are also available through JMS by the use of object properties.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>JMS Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBodyInputStream(InputStream)</td>
<td>Set the InputStream used to read a message body when sending it.</td>
<td>JMS_AMQ_InputStream</td>
</tr>
<tr>
<td>setOutputStream(OutputStream)</td>
<td>Set the OutputStream that will receive the body of a message. This method does not block.</td>
<td>JMS_AMQ_OutputStream</td>
</tr>
<tr>
<td>saveOutputStream(OutputStream)</td>
<td>Save the body of the message to the OutputStream. It will block until the entire content is transferred to the OutputStream.</td>
<td>JMS_AMQ_SaveStream</td>
</tr>
</tbody>
</table>

To set the output stream when receiving a core message:

```java
ClientMessage msg = consumer.receive(...);
// This will block here until the stream was transferred
msg.saveOutputStream(someOutputStream);

ClientMessage msg2 = consumer.receive(...);
// This will not wait the transfer to finish
msg2.setOutputStream(someOtherOutputStream);
```

Set the input stream when sending a core message:

```java
ClientMessage msg = session.createMessage();
msg.setInputStream(dataInputStream);
```

Notice also that for messages with more than 2GiB the getBodySize() will return invalid values since this is an integer (which is also exposed to the JMS API). On those cases you can use the message property _AMQ_LARGE_SIZE.

58.4.2. Streaming over JMS

When using JMS, Apache ActiveMQ Artemis maps the streaming methods on the core API (see `ClientMessage` API table above) by setting object properties. You can use the method `Message.setObjectProperty` to set the input and output streams.

The `InputStream` can be defined through the JMS Object Property `JMS_AMQ_InputStream` on messages being sent:
BytesMessage message = session.createBytesMessage();

FileInputStream fileInputStream = new FileInputStream(fileInput);

BufferedInputStream bufferedInput = new BufferedInputStream(fileInputStream);

message.setObjectProperty("JMS_AMQ_InputStream", bufferedInput);

someProducer.send(message);

The OutputStream can be set through the JMS Object Property JMS_AMQ_SaveStream on messages being received in a blocking way.

BytesMessage messageReceived = (BytesMessage)messageConsumer.receive(120000);

File outputFile = new File("huge_message_received.dat");

FileOutputStream fileOutputStream = new FileOutputStream(outputFile);

BufferedOutputStream bufferedOutput = new BufferedOutputStream(fileOutputStream);

// This will block until the entire content is saved on disk
messageReceived.setObjectProperty("JMS_AMQ_SaveStream", bufferedOutput);

Setting the OutputStream could also be done in a non blocking way using the property JMS_AMQ_OutputStream.

// This won't wait the stream to finish. You need to keep the consumer active.
messageReceived.setObjectProperty("JMS_AMQ_OutputStream", bufferedOutput);

When using JMS, Streaming large messages are only supported on StreamMessage and BytesMessage.

58.4.3. Streaming Alternative on Core Protocol

If you choose not to use the InputStream or OutputStream capability of Apache ActiveMQ Artemis You could still access the data directly in an alternative fashion.

On the Core API just get the bytes of the body as you normally would.

ClientMessage msg = consumer.receive();

byte[] bytes = new byte[1024];
for (int i = 0; i < msg.getBodySize(); i += bytes.length)
{
    msg.getBody().readBytes(bytes);
If using JMS API, BytesMessage and StreamMessage also supports it transparently.

```java
BytesMessage rm = (BytesMessage)cons.receive(10000);

byte data[] = new byte[1024];

for (int i = 0; i < rm.getBodyLength(); i += 1024)
{
  int numberOfBytes = rm.readBytes(data);
  // Do whatever you want with the data
}
```

### 58.5. Configuring AMQP Acceptor

You can configure the property `amqpMinLargeMessageSize` at the acceptor.

The default value is 102400 (100KBytes).

Setting it to -1 will disable large message support.

> setting amqpMinLargeMessageSize to -1, your AMQP message might be stored as a Core Large Message if the size of the message does not fit into the journal. This is the former semantic of the broker and it is kept this way for compatibility reasons.

```xml
<acceptors>
  <!-- AMQP Acceptor. Listens on default AMQP port for AMQP traffic.--> 
  <acceptor name="amqp">tcp://0.0.0.0:5672; ....
  amqpMinLargeMessageSize=102400; .... </acceptor>
</acceptors>
```

### 58.6. Large message example

Please see the Large Message Example which shows how large messages are configured and used with JMS.
Chapter 59. Paging

Apache ActiveMQ Artemis transparently supports huge queues containing millions of messages while the server is running with limited memory.

In such a situation it’s not possible to store all of the queues in memory at any one time, so Apache ActiveMQ Artemis transparently pages messages into and out of memory as they are needed, thus allowing massive queues with a low memory footprint.

Apache ActiveMQ Artemis will start paging messages to disk, when the size of all messages in memory for an address exceeds a configured maximum size.

The default configuration from Artemis has destinations with paging.

59.1. Page Files

Messages are stored per address on the file system. Each address has an individual folder where messages are stored in multiple files (page files). Each file will contain messages up to a max configured size (page-size-bytes). The system will navigate the files as needed, and it will remove the page file as soon as all the messages are acknowledged up to that point.

Browsers will read through the page-cursor system.

Consumers with selectors will also navigate through the page-files and it will ignore messages that don’t match the criteria.

When you have a queue, and consumers filtering the queue with a very restrictive selector you may get into a situation where you won’t be able to read more data from paging until you consume messages from the queue.

Example: in one consumer you make a selector as ‘color="red"’ but you only have one color red 1 millions messages after blue, you won’t be able to consume red until you consume blue ones.

This is different to browsing as we will "browse" the entire queue looking for messages and while we "depage" messages while feeding the queue.

59.1.1. Configuration

You can configure the location of the paging folder in broker.xml.

- paging-directory Where page files are stored. Apache ActiveMQ Artemis will create one folder for each address being paged under this configured location. Default is data/paging.

59.2. Paging Mode

As soon as messages delivered to an address exceed the configured size, that address alone goes into page mode. If max-size-bytes == 0 or max-size-messages == 0, an address will always use paging
to route messages.

Paging is done individually per address. If you configure a max-size-bytes or max-messages for an address, that means each matching address will have a maximum size that you specified. It DOES NOT mean that the total overall size of all matching addresses is limited to max-size-bytes. Use global-max-size or global-max-messages for that!

59.2.1. Configuration

Configuration is done at the address settings in broker.xml.

```xml
<address-settings>
  <address-setting match="jms.someaddress">
    <max-size-bytes>104857600</max-size-bytes>
    <max-size-messages>1000</max-size-messages>
    <page-size-bytes>10485760</page-size-bytes>
    <address-full-policy>PAGE</address-full-policy>
    <page-limit-bytes>10G</page-limit-bytes>
    <page-limit-messages>1000000</page-limit-messages>
    <page-full-policy>FAIL</page-full-policy>
  </address-setting>
</address-settings>
```

The management-address settings cannot be changed or overridden i.e., management messages aren't allowed to page/block/fail and are considered an internal broker management mechanism. The memory occupation of the management-address is not considered while evaluating if global-max-size is hit and can't cause other non-management addresses to trigger a configured address-full-policy.

This is the list of available parameters on the address settings.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>max-size-bytes</td>
<td>What's the max memory the address could have before entering on page mode.</td>
<td>-1 (disabled)</td>
</tr>
<tr>
<td>max-size-messages</td>
<td>The max number of messages the address could have before entering on page mode.</td>
<td>-1 (disabled)</td>
</tr>
<tr>
<td>page-size-bytes</td>
<td>The size of each page file used on the paging system</td>
<td>10MB</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>address-full-policy</td>
<td>This must be set to <strong>PAGE</strong> for paging to enable. If the value is <strong>PAGE</strong> then further messages will be paged to disk. If the value is <strong>DROP</strong> then further messages will be silently dropped. If the value is <strong>FAIL</strong> then the messages will be dropped and the client message producers will receive an exception. If the value is <strong>BLOCK</strong> then client message producers will block when they try and send further messages.</td>
<td><strong>PAGE</strong></td>
</tr>
<tr>
<td>max-read-page-messages</td>
<td>Maximum number of paged messages that the broker can read into memory per-queue. The default value is -1, which means that no limit applies.</td>
<td>-1 (disabled)</td>
</tr>
<tr>
<td>max-read-page-bytes</td>
<td>Maximum memory, in bytes, that can be used to read paged messages into memory per-queue. When applying this limit, the broker takes into account both messages that are currently delivering and messages that are ready to be delivered to consumers. The default value is (2 \times \text{page-size-bytes}) (usually being 20 MB). If consumers are slow to acknowledge messages, you can increase the default value to ensure that the memory is not consumed by messages pending acknowledgment, which can starve the broker of messages.</td>
<td>(2 \times \text{page-size-bytes})</td>
</tr>
<tr>
<td>prefetch-page-messages</td>
<td>Number of paged messages that the broker can read from disk into memory per-queue. The default value is taken from max-read-page-messages, usually at -1, which means that no limit applies.</td>
<td>max-read-page-messages</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>prefetch-page-bytes</td>
<td>Number of paged messages that the broker can read from disk into memory per-queue. The default value is taken from max-read-page-messages, usually at -1, which means that no limit applies.</td>
<td>If not defined, max-read-page-bytes</td>
</tr>
<tr>
<td>page-limit-bytes</td>
<td>After entering page mode, how much data would the system allow incoming. Notice this will be internally converted as number of pages.</td>
<td></td>
</tr>
<tr>
<td>page-limit-messages</td>
<td>After entering page mode, how many messages would the system allow incoming on paging.</td>
<td></td>
</tr>
<tr>
<td>page-full-policy</td>
<td>Valid results are DROP or FAIL. This tells what to do if the system is reaching page-limit-bytes or page-limit-messages after paging</td>
<td></td>
</tr>
</tbody>
</table>

When using the JDBC storage, the effective page-size-bytes used is limited to jdbc-max-page-size-bytes, configured in the JDBC storage section.

59.2.2. max-size-bytes and max-size-messages simultaneous usage

It is possible to define max-size-messages (as the maximum number of messages) and max-messages-size (as the max number of estimated memory used by the address) concurrently. The configured policy will start based on the first value to reach its mark.

**Maximum read from page**

max-read-page-messages, max-read-page-bytes, prefetch-page-messages and prefetch-page-bytes are used to control reading from paged file into the Queue. The broker will add messages as long as all these limits are satisfied.

If all these values are set to -1 the broker will keep reading messages as long as the consumer is reaching for more messages. However this would keep the broker unprotected from consumers allocating huge transactions or consumers that don't have flow control enabled.

59.3. Global Max Size

Beyond the max-size-bytes on the address you can also set the global-max-size on the main configuration. If you set max-size-bytes = -1 on paging the global-max-size can still be used.
59.4. Global Max Messages

You can also specify `global-max-messages` on the main configuration, specifying how many messages the system would accept before entering into the configured full policy mode configured.

When you have more messages than what is configured `global-max-size` any new produced message will make that destination to go through its paging policy.

`global-max-size` is calculated as half of the max memory available to the Java Virtual Machine, unless specified on the `broker.xml` configuration.

By default `global-max-messages = -1` meaning it's disabled.

59.5. Dropping messages

Instead of paging messages when the max size is reached, an address can also be configured to just drop messages when the address is full.

To do this just set the `address-full-policy` to `DROP` in the address settings.

59.6. Dropping messages and throwing an exception to producers

Instead of paging messages when the max size is reached, an address can also be configured to drop messages and also throw an exception on the client-side when the address is full.

To do this just set the `address-full-policy` to `FAIL` in the address settings.

59.7. Blocking producers

Instead of paging messages when the max size is reached, an address can also be configured to block producers from sending further messages when the address is full, thus preventing the memory being exhausted on the server.

When memory is freed up on the server, producers will automatically unblock and be able to continue sending.

To do this just set the `address-full-policy` to `BLOCK` in the address settings.

In the default configuration, all addresses are configured to block producers after 10 MiB of data are in the address.

59.8. Caution with Addresses with Multiple Multicast Queues

When a message is routed to an address that has multiple multicast queues bound to it, e.g. a JMS subscription in a Topic, there is only 1 copy of the message in memory. Each queue only deals with
a reference to this. Because of this the memory is only freed up once all queues referencing the message have delivered it.

If you have a single lazy subscription, the entire address will suffer IO performance hit as all the queues will have messages being sent through an extra storage on the paging system.

For example:

- An address has 10 multicast queues
- One of the queues does not deliver its messages (maybe because of a slow consumer).
- Messages continually arrive at the address and paging is started.
- The other 9 queues are empty even though messages have been sent.

In this example all the other 9 queues will be consuming messages from the page system. This may cause performance issues if this is an undesirable state.

### 59.9. Monitoring Disk

The broker can be configured to perform scans on the disk to determine if disk is beyond a configured limit. Since the disk is a critical piece of infrastructure for data integrity the broker will automatically shut itself down if it runs out of disk space. Configuring a limit allows the broker to enforce flow control on clients sending messages to the broker so that the disk never fills up completely.

- If the protocol used to send the messages doesn’t support flow control (e.g. STOMP) then an exception will be thrown and the connection for the client will be dropped so that it can no longer send messages and consume disk space.

#### 59.9.1. Max Disk Usage

A limit on the *maximum* disk space used can be configured through `max-disk-usage`. This is the percentage of disk used. For example, if the disk’s capacity was 500GiB and `max-disk-usage` was 50 then the broker would start blocking producers once 250GiB of disk space was used.

#### 59.9.2. Minimum Disk Free

A limit on the *minimum* disk space free can be configured through `min-disk-free`. This is specific amount and not a percentage like with `max-disk-usage`. For example, if the disk’s capacity was 500GiB and `min-disk-free` was 100GiB then the broker would start blocking producers once 400GiB of disk space was used.

If both `max-disk-usage` and `min-disk-free` are configured then `min-disk-free` will take priority.
59.10. Page Sync Timeout

The pages are synced periodically and the sync period is configured through \texttt{page-sync-timeout} in nanoseconds. When using NIO journal, by default has the same value of \texttt{journal-buffer-timeout}. When using ASYNCIO, the default should be 333333.

59.11. Memory usage from Paged Messages.

The system should keep at least one paged file in memory caching ahead reading messages. Also every active subscription could keep one paged file in memory. So, if your system has too many queues it is recommended to minimize the page-size.

59.12. Page Limits and Page Full Policy

Since version 2.28.0 is possible to configure limits on how much data is paged. This is to avoid a single destination using the entire disk in case their consumers are gone.

You can configure either \texttt{page-limit-bytes} or \texttt{page-limit-messages}, along with \texttt{page-full-policy} on the address settings limiting how much data will be recorded in paging.

If you configure \texttt{page-full-policy} as DROP, messages will be simplify dropped while the clients will not get any exceptions, while if you configured FAIL the producers will receive a JMS Exception for the error condition.

59.13. Example

See the Paging Example which shows how to use paging with Apache ActiveMQ Artemis.
Chapter 60. Duplicate Message Detection

Apache ActiveMQ Artemis includes powerful automatic duplicate message detection, filtering out duplicate messages without you having to code your own fiddly duplicate detection logic at the application level. This chapter will explain what duplicate detection is, how Apache ActiveMQ Artemis uses it and how and where to configure it.

When sending messages from a client to a server, or indeed from a server to another server, if the target server or connection fails sometime after sending the message, but before the sender receives a response that the send (or commit) was processed successfully then the sender cannot know for sure if the message was sent successfully to the address.

If the target server or connection failed after the send was received and processed but before the response was sent back then the message will have been sent to the address successfully, but if the target server or connection failed before the send was received and finished processing then it will not have been sent to the address successfully. From the senders point of view it’s not possible to distinguish these two cases.

When the server recovers this leaves the client in a difficult situation. It knows the target server failed, but it does not know if the last message reached its destination ok. If it decides to resend the last message, then that could result in a duplicate message being sent to the address. If each message was an order or a trade then this could result in the order being fulfilled twice or the trade being double booked. This is clearly not a desirable situation.

Sending the message(s) in a transaction does not help out either. If the server or connection fails while the transaction commit is being processed it is also indeterminate whether the transaction was successfully committed or not!

To solve these issues Apache ActiveMQ Artemis provides automatic duplicate messages detection for messages sent to addresses.

60.1. Using Duplicate Detection for Message Sending

Enabling duplicate message detection for sent messages is simple: you just need to set a special property on the message to a unique value. You can create the value however you like, as long as it is unique. When the target server receives the message it will check if that property is set, if it is, then it will check in its in memory cache if it has already received a message with that value of the header. If it has received a message with the same value before then it will ignore the message.

Using duplicate detection to move messages between nodes can give you the same once and only once delivery guarantees as if you were using an XA transaction to consume messages from source and send them to the target, but with less overhead and much easier configuration than using XA.

If you’re sending messages in a transaction then you don’t have to set the property for every message you send in that transaction, you only need to set it once in the transaction. If the server detects a duplicate message for any message in the transaction, then it will ignore the entire transaction.
The name of the property that you set is given by the value of org.apache.activemq.artemis.api.core.Message.HDR_DUPLICATE_DETECTION_ID, which is _AMQ_DUPL_ID_.

The value of the property can be of type `byte[]` or `SimpleString` if you're using the core API. If you're using JMS it must be a `String`, and its value should be unique. An easy way of generating a unique id is by generating a UUID.

Here's an example of setting the property using the core API:

```java
... 
ClientMessage message = session.createMessage(true);
SimpleString myUniqueID = "This is my unique id"; // Could use a UUID for this
message.setStringProperty(HDR_DUPLICATE_DETECTION_ID, myUniqueID);
... 
```

And here's an example using the JMS API:

```java
... 
Message jmsMessage = session.createMessage();
String myUniqueID = "This is my unique id"; // Could use a UUID for this
message.setStringProperty(HDR_DUPLICATE_DETECTION_ID.toString(), myUniqueID);
... 
```

### 60.2. Configuring the Duplicate ID Cache

The server maintains caches of received values of the org.apache.activemq.artemis.core.message.impl.HDR_DUPLICATE_DETECTION_ID property sent to each address. Each address has its own distinct cache.

The cache is a circular fixed size cache. If the cache has a maximum size of n elements, then the n + 1th id stored will overwrite the 0th element in the cache.

The maximum size of the cache is configured by the parameter `id-cache-size` in broker.xml, the default value is 20000 elements.

To implement an address-specific `id-cache-size`, you can add to the corresponding address-settings section in broker.xml. Specify the desired `id-cache-size` value for the particular address. When a message is sent to an address with a specific `id-cache-size` configured, it will take precedence over the global `id-cache-size` value, allowing for greater flexibility and optimization of duplicate ID caching.

The caches can also be configured to persist to disk or not. This is configured by the parameter
**60.3. Duplicate Detection and Bridges**

Core bridges can be configured to automatically add a unique duplicate id value (if there isn’t already one in the message) before forwarding the message to its target. This ensures that if the target server crashes or the connection is interrupted and the bridge resends the message, then if it has already been received by the target server, it will be ignored.

To configure a core bridge to add the duplicate id header, simply set the `use-duplicate-detection` to `true` when configuring a bridge in `broker.xml`.

The default value for this parameter is `true`.

For more information on core bridges and how to configure them, please see [Core Bridges](#).

**60.4. Duplicate Detection and Cluster Connections**

Cluster connections internally use core bridges to move messages reliably between nodes of the cluster. Consequently they can also be configured to insert the duplicate id header for each message they move using their internal bridges.

To configure a cluster connection to add the duplicate id header, simply set the `use-duplicate-detection` to `true` when configuring a cluster connection in `broker.xml`.

The default value for this parameter is `true`.

For more information on cluster connections and how to configure them, please see [Clusters](#).
Chapter 61. Message Redelivery and Undelivered Messages

Messages can be delivered unsuccessfully (e.g. if the transacted session used to consume them is rolled back). Such a message goes back to its queue ready to be redelivered. However, this means it is possible for a message to be delivered again and again without success thus remaining in the queue indefinitely, clogging the system.

There are 2 ways to deal with these undelivered messages:

- **Delayed redelivery.**
  It is possible to delay messages redelivery. This gives the client some time to recover from any transient failures and to prevent overloading its network or CPU resources.

- **Dead Letter Address.**
  It is also possible to configure a dead letter address so that after a specified number of unsuccessful deliveries, messages are removed from their queue and sent to the dead letter address. These messages will not be delivered again from this queue.

Both options can be combined for maximum flexibility.

61.1. Delayed Redelivery

Delaying redelivery can often be useful in cases where clients regularly fail or rollback. Without a delayed redelivery, the system can get into a “thrashing” state, with delivery being attempted, the client rolling back, and delivery being re-attempted ad infinitum in quick succession, consuming valuable CPU and network resources.

#Persist Redelivery

Two Journal update records are stored every time a redelivery happens. One for the number of deliveries that happened, and one in case a scheduled redelivery is being used.

It is recommended to keep max-redelivery-records=1 in situations where you are operating with very short redelivery delays as you will be creating unnecessary records on the journal.

61.1.1. Configuring Delayed Redelivery

Delayed redelivery is defined in the address-setting configuration:

```xml
<!-- delay redelivery of messages for 5s -->
<address-setting match="exampleQueue">
  <!-- default is 1.0 -->
  <redelivery-delay-multiplier>1.5</redelivery-delay-multiplier>
  <!-- default is 0 (no delay) -->
  <redelivery-delay>5000</redelivery-delay>
</address-setting>
```
If a `redelivery-delay` is specified, Apache ActiveMQ Artemis will wait this delay before redelivering the messages.

By default, there is no redelivery delay (`redelivery-delay` is set to 0).

Other subsequent messages will be delivery regularly, only the cancelled message will be sent asynchronously back to the queue after the delay.

You can specify a multiplier (the `redelivery-delay-multiplier`) that will take effect on top of the `redelivery-delay`. Each time a message is redelivered the delay period will be equal to the previous delay * `redelivery-delay-multiplier`. A `max-redelivery-delay` can be set to prevent the delay from becoming too large. The `max-redelivery-delay` is defaulted to `redelivery-delay * 10`.

**Example:**

- `redelivery-delay=5000`, `redelivery-delay-multiplier=2`, `max-redelivery-delay=15000`, `redelivery-collision-avoidance-factor=0.0`
  1. Delivery Attempt 1. (Unsuccessful)
  2. Wait Delay Period: 5000
  3. Delivery Attempt 2. (Unsuccessful)
  4. Wait Delay Period: 10000 // (5000 * 2) < max-delay-period. Use 10000
  5. Delivery Attempt 3: (Unsuccessful)
  6. Wait Delay Period: 15000 // (10000 * 2) > max-delay-period: Use max-delay-delivery

Address wildcards can be used to configure redelivery delay for a set of addresses (see [Understanding the Wildcard Syntax](#)), so you don’t have to specify redelivery delay individually for each address.

The `redelivery-delay` can be also be modified by configuring the `redelivery-collision-avoidance-factor`. This factor will be made either positive or negative at random to control whether the ultimate value will increase or decrease the `redelivery-delay`. Then it's multiplied by a random number between 0.0 and 1.0. This result is then multiplied by the `redelivery-delay` and then added to the `redelivery-delay` to arrive at the final value.

The algorithm may sound complicated but the bottom line is quite simple: the larger `redelivery-collision-avoidance-factor` you choose the larger the variance of the `redelivery-delay` will be. The `redelivery-collision-avoidance-factor` must be between 0.0 and 1.0.

**Example:**

- `redelivery-delay=1000`, `redelivery-delay-multiplier=1`, `max-redelivery-delay=15000`, `redelivery-collision-avoidance-factor=0.5`, (bold values chosen using java.util.Random)
1. Delivery Attempt 1. (Unsuccessful)
2. Wait Delay Period: $875 \div 1000 + (1000 \times ((0.5 \times -1) \times 0.25)$
3. Delivery Attempt 2. (Unsuccessful)
4. Wait Delay Period: $1375 \div 1000 + (1000 \times ((0.5 \times 1) \times 0.75)$
5. Delivery Attempt 3: (Unsuccessful)
6. Wait Delay Period: $975 \div 1000 + (1000 \times ((0.5 \times -1) \times 0.05)$

This feature can be particularly useful in environments where there are multiple consumers on the same queue all interacting transactionally with the same external system (e.g. a database). If there is overlapping data in messages which are consumed concurrently then one transaction can succeed while all the rest fail. If those failed messages are redelivered at the same time then this process where one consumer succeeds and the rest fail will continue. By randomly padding the redelivery-delay by a small, configurable amount these redelivery "collisions" can be avoided.

### 61.1.2. Example

See the examples chapter for an example which shows how delayed redelivery is configured and used with JMS.

### 61.2. Dead Letter Addresses

To prevent a client infinitely receiving the same undelivered message (regardless of what is causing the unsuccessful deliveries), messaging systems define **dead letter addresses**: after a specified unsuccessful delivery attempts, the message is removed from its queue and sent to a dead letter address.

Any such messages can then be diverted to queue(s) where they can later be perused by the system administrator for action to be taken.

Apache ActiveMQ Artemis's addresses can be assigned a dead letter address. Once the messages have been unsuccessfully delivered for a given number of attempts, they are removed from their queue and sent to the relevant dead letter address. These **dead letter** messages can later be consumed from the dead letter address for further inspection.

#### 61.2.1. Configuring Dead Letter Addresses

Dead letter address is defined in the address-setting configuration:

```xml
<!-- undelivered messages in exampleQueue will be sent to the dead letter address deadLetterQueue after 3 unsuccessful delivery attempts -->
<address-setting match="exampleQueue">
  <dead-letter-address>deadLetterAddress</dead-letter-address>
  <max-delivery-attempts>3</max-delivery-attempts>
</address-setting>
```

If a **dead-letter-address** is not specified, messages will be removed after **max-delivery-attempts**
unsuccessful attempts.

By default, messages are redelivered 10 times at the maximum. Set \texttt{max-delivery-attempts} to -1 for infinite redeliveries.

A \textbf{dead letter address} can be set globally for a set of matching addresses and you can set \texttt{max-delivery-attempts} to -1 for a specific address setting to allow infinite redeliveries only for this address.

Address wildcards can be used to configure dead letter settings for a set of addresses (see \texttt{Understanding the Wildcard Syntax}).

### 61.2.2. Dead Letter Properties

Dead letter messages get \textbf{special properties}.

### 61.2.3. Automatically Creating Dead Letter Resources

It's common to segregate undelivered messages by their original address. For example, a message sent to the \texttt{stocks} address that couldn't be delivered for some reason might be ultimately routed to the \texttt{DLQ.stocks} queue, and likewise a message sent to the \texttt{orders} address that couldn't be delivered might be routed to the \texttt{DLQ.orders} queue.

Using this pattern can make it easy to track and administrate undelivered messages. However, it can pose a challenge in environments which predominantly use auto-created addresses and queues. Typically administrators in those environments don't want to manually create an \texttt{address-setting} to configure the \texttt{dead-letter-address} much less the actual \texttt{address} and \texttt{queue} to hold the undelivered messages.

The solution to this problem is to set the \texttt{auto-create-dead-letter-resources address-setting} to \texttt{true} (it's \texttt{false} by default) so that the broker will create the \texttt{address} and \texttt{queue} to deal with the undelivered messages automatically. The \texttt{address} created will be the one defined by the \texttt{dead-letter-address}. A MULTICAST queue will be created on that \texttt{address}. It will be named by the \texttt{address} to which the message was previously sent, and it will have a filter defined using the property \texttt{_AMQ_ORIG_ADDRESS} so that it will only receive messages sent to the relevant \texttt{address}. The \texttt{queue} name can be configured with a prefix and suffix. See the relevant settings in the table below:

<table>
<thead>
<tr>
<th>address-setting</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>dead-letter-queue-prefix</td>
<td>DLQ.</td>
</tr>
<tr>
<td>dead-letter-queue-suffix</td>
<td>(empty string)</td>
</tr>
</tbody>
</table>

Here is an example configuration:

```
<address-setting match="#">
  <dead-letter-address>DLA</dead-letter-address>
  <max-delivery-attempts>3</max-delivery-attempts>
  <auto-create-dead-letter-resources>true</auto-create-dead-letter-resources>
  <dead-letter-queue-prefix></dead-letter-queue-prefix>
  <!-- override the default -->
  <dead-letter-queue-suffix>.DLQ</dead-letter-queue-suffix>
</address-setting>
```
The queue holding the undeliverable messages can be accessed directly either by using the queue’s name by itself (e.g. when using the core client) or by using the fully qualified queue name (e.g. when using a JMS client) just like any other queue. Also, note that the queue is auto-created which means it will be auto-deleted as per the relevant address-settings.

61.2.4. Example

See: Dead Letter section of the Examples for an example that shows how dead letter resources can be statically configured and used with JMS.

61.3. Delivery Count Persistence

In normal use, Apache ActiveMQ Artemis does not update delivery count persistently until a message is rolled back (i.e. the delivery count is not updated before the message is delivered to the consumer). In most messaging use cases, the messages are consumed, acknowledged and forgotten as soon as they are consumed. In these cases, updating the delivery count persistently before delivering the message would add an extra persistent step for each message delivered, implying a significant performance penalty.

However, if the delivery count is not updated persistently before the message delivery happens, in the event of a server crash, messages might have been delivered but that will not have been reflected in the delivery count. During the recovery phase, the server will not have knowledge of that and will deliver the message with redelivered set to false while it should be true.

As this behavior breaks strict JMS semantics, Apache ActiveMQ Artemis allows to persist delivery count before message delivery but this feature is disabled by default due to performance implications.

To enable it, set persist-delivery-count-before-delivery to true in broker.xml:

```xml
<persist-delivery-count-before-delivery>true</persist-delivery-count-before-delivery>
```
Chapter 62. Persistence

Apache ActiveMQ Artemis ships with two persistence options. The file journal which is highly optimized for the messaging use case and gives great performance, and also the JDBC Store, which uses JDBC to connect to a database of your choice.

62.1. File Journal (Default)

The file journal is an *append only* journal. It consists of a set of files on disk. Each file is pre-created to a fixed size and initially filled with padding. As operations are performed on the server, e.g. add message, update message, delete message, records are appended to the journal. When one journal file is full we move to the next one.

Because records are only appended, i.e. added to the end of the journal we minimise disk head movement, i.e. we minimise random access operations which is typically the slowest operation on a disk.

Making the file size configurable means that an optimal size can be chosen, i.e. making each file fit on a disk cylinder. Modern disk topologies are complex and we are not in control over which cylinder(s) the file is mapped onto so this is not an exact science. But by minimising the number of disk cylinders the file is using, we can minimise the amount of disk head movement, since an entire disk cylinder is accessible simply by the disk rotating - the head does not have to move.

As delete records are added to the journal, Apache ActiveMQ Artemis has a sophisticated file garbage collection algorithm which can determine if a particular journal file is needed any more - i.e. has all its data been deleted in the same or other files. If so, the file can be reclaimed and re-used.

Apache ActiveMQ Artemis also has a compaction algorithm which removes dead space from the journal and compresses up the data so it takes up less files on disk.

The journal also fully supports transactional operation if required, supporting both local and XA transactions.

The majority of the journal is written in Java, however we abstract out the interaction with the actual file system to allow different pluggable implementations. Apache ActiveMQ Artemis ships with two implementations:

62.1.1. Journal and Data Retention

ActiveMQ Artemis provides a way to store and replay historical data. Refer to Data Retention chapter for more information.

62.1.2. Java NIO

The first implementation uses standard Java NIO to interface with the file system. This provides extremely good performance and runs on any platform where there's a Java 6+ runtime.
62.1.3. Linux Asynchronous IO

The second implementation uses a thin native code wrapper to talk to the Linux asynchronous IO library (AIO). With AIO, Apache ActiveMQ Artemis will be called back when the data has made it to disk, allowing us to avoid explicit syncs altogether and simply send back confirmation of completion when AIO informs us that the data has been persisted.

Using AIO will typically provide even better performance than using Java NIO.

This journal option is only available when running Linux kernel 2.6 or later and after having installed libaio (if it’s not already installed). For instructions on how to install libaio please see Installing AIO section.

Also, please note that AIO will only work with the following file systems: ext2, ext3, ext4, jfs, xfs and NFSV4.

For more information on libaio please see lib AIO.

libaio is part of the kernel project.

62.1.4. Memory mapped

The third implementation uses a file-backed READ_WRITE memory mapping against the OS page cache to interface with the file system.

This provides extremely good performance (especially under strictly process failure durability requirements), almost zero copy (actually is the kernel page cache) and zero garbage (from the Java HEAP perspective) operations and runs on any platform where there’s a Java 4+ runtime.

Under power failure durability requirements it will perform at least on par with the NIO journal with the only exception of Linux OS with kernel less or equals 2.6, in which the msync) implementation necessary to ensure durable writes was different (and slower) from the fsync used is case of NIO journal.

It benefits by the configuration of OS huge pages, in particular when is used a big number of journal files and sizing them as multiple of the OS page size in bytes.

62.1.5. Standard Files

The standard Apache ActiveMQ Artemis core server uses two instances of the journal:

• Bindings journal.

This journal is used to store bindings related data. That includes the set of queues that are deployed on the server and their attributes. It also stores data such as id sequence counters.

The bindings journal is always a NIO journal as it is typically low throughput compared to the message journal.

The files on this journal are prefixed as activemq-bindings. Each file has a bindings extension. File size is 1048576, and it is located at the bindings folder.
• Message journal.

This journal instance stores all message related data, including the message themselves and also duplicate-id caches.

By default Apache ActiveMQ Artemis will try and use an AIO journal. If AIO is not available, e.g. the platform is not Linux with the correct kernel version or AIO has not been installed then it will automatically fall back to using Java NIO which is available on any Java platform.

The files on this journal are prefixed as activemq-data. Each file has an amq extension. File size is by the default 10485760 (configurable), and it is located at the journal folder.

For large messages, Apache ActiveMQ Artemis persists them outside the message journal. This is discussed in Large Messages.

Apache ActiveMQ Artemis can also be configured to page messages to disk in low memory situations. This is discussed in Paging.

If no persistence is required at all, Apache ActiveMQ Artemis can also be configured not to persist any data at all to storage as discussed in the Configuring the broker for Zero Persistence section.

Configuring the bindings journal

The bindings journal is configured using the following attributes in broker.xml

bindings-directory

This is the directory in which the bindings journal lives. The default value is data/bindings.

create-bindings-dir

If this is set to true then the bindings directory will be automatically created at the location specified in bindings-directory if it does not already exist. The default value is true

Configuring the jms journal

The jms config shares its configuration with the bindings journal.

Configuring the message journal

The message journal is configured using the following attributes in broker.xml

journal-directory

This is the directory in which the message journal lives. The default value is data/journal.

For the best performance, we recommend the journal is located on its own physical volume in order to minimise disk head movement. If the journal is on a volume which is shared with other processes which might be writing other files (e.g. bindings journal, database, or transaction coordinator) then the disk head may well be moving rapidly between these files as it writes them, thus drastically reducing performance.

When the message journal is stored on a SAN we recommend each journal instance that is stored on the SAN is given its own LUN (logical unit).
node-manager-lock-directory

This is the directory in which the node manager file lock lives. By default has the same value of journal-directory.

This is useful when the message journal is on a SAN and is being used a Shared Store HA policy with the broker instances on the same physical machine.

create-journal-dir

If this is set to true then the journal directory will be automatically created at the location specified in journal-directory if it does not already exist. The default value is true.

journal-type

Valid values are NIO, ASYNCIO or MAPPED.

Choosing NIO chooses the Java NIO journal. Choosing ASYNCIO chooses the Linux asynchronous IO journal. If you choose ASYNCIO but are not running Linux or you do not have libaio installed then Apache ActiveMQ Artemis will detect this and automatically fall back to using NIO. Choosing MAPPED chooses the Java Memory Mapped journal.

journal-sync-transactional

If this is set to true then Apache ActiveMQ Artemis will make sure all transaction data is flushed to disk on transaction boundaries (commit, prepare and rollback). The default value is true.

journal-sync-non-transactional

If this is set to true then Apache ActiveMQ Artemis will make sure non transactional message data (sends and acknowledgements) are flushed to disk each time. The default value for this is true.

journal-file-size

The size of each journal file in bytes. The default value for this is 10485760 bytes (10MiB).

journal-min-files

The minimum number of files the journal will maintain. When Apache ActiveMQ Artemis starts and there is no initial message data, Apache ActiveMQ Artemis will pre-create journal-min-files number of files.

Creating journal files and filling them with padding is a fairly expensive operation and we want to minimise doing this at run-time as files get filled. By pre-creating files, as one is filled the journal can immediately resume with the next one without pausing to create it.

Depending on how much data you expect your queues to contain at steady state you should tune this number of files to match that total amount of data.

journal-pool-files

The system will create as many files as needed however when reclaiming files it will shrink back to the journal-pool-files.

The default to this parameter is -1, meaning it will never delete files on the journal once created.
Notice that the system can't grow infinitely as you are still required to use paging for destinations that can grow indefinitely.

Notice: in case you get too many files you can use compacting.

**journal-max-io**

Write requests are queued up before being submitted to the system for execution. This parameter controls the maximum number of write requests that can be in the IO queue at any one time. If the queue becomes full then writes will block until space is freed up.

When using NIO, this value should always be equal to 1

When using ASYNCIO, the default should be 500.

The system maintains different defaults for this parameter depending on whether it's NIO or ASYNCIO (default for NIO is 1, default for ASYNCIO is 500)

There is a limit and the total max ASYNCIO can't be higher than what is configured at the OS level (/proc/sys/fs/aio-max-nr) usually at 65536.

**journal-buffer-timeout**

Instead of flushing on every write that requires a flush, we maintain an internal buffer, and flush the entire buffer either when it is full, or when a timeout expires, whichever is sooner. This is used for both NIO and ASYNCIO and allows the system to scale better with many concurrent writes that require flushing.

This parameter controls the timeout at which the buffer will be flushed if it hasn't filled already. ASYNCIO can typically cope with a higher flush rate than NIO, so the system maintains different defaults for both NIO and ASYNCIO (default for NIO is 3333333 nanoseconds - 300 times per second, default for ASYNCIO is 500000 nanoseconds - ie. 2000 times per second).

Setting this property to 0 will disable the internal buffer and writes will be directly written to the journal file immediately.

By increasing the timeout, you may be able to increase system throughput at the expense of latency, the default parameters are chosen to give a reasonable balance between throughput and latency.

**journal-buffer-size**

The size of the timed buffer on ASYNCIO. The default value is 490KiB.

**journal-compact-min-files**

The minimal number of files before we can consider compacting the journal. The compacting algorithm won't start until you have at least journal-compact-min-files

Setting this to 0 will disable the feature to compact completely. This could be dangerous though as the journal could grow indefinitely. Use it wisely!

The default for this parameter is 10
**journal-compact-percentage**

The threshold to start compacting. When less than this percentage of journal space is considered live data, we start compacting. Note also that compacting won’t kick in until you have at least **journal-compact-min-files** data files on the journal.

The default for this parameter is **30**.

**journal-lock-acquisition-timeout**

How long to wait (in milliseconds) to acquire a file lock on the journal before giving up.

The default for this parameter is **-1** (i.e. indefinite).

**journal-datasync**

This will disable the use of fdatasync on journal writes. When enabled it ensures full power failure durability, otherwise process failure durability on journal writes (OS guaranteed). This is particular effective for **NIO** and **MAPPED** journals, which rely on **fsync/msync** to force write changes to disk.

Default is **true**.

### Note on disabling journal-datasync

Any modern OS guarantees that on process failures (i.e. crash) all the uncommitted changes to the page cache will be flushed to the file system, maintaining coherence between subsequent operations against the same pages and ensuring that no data will be lost. The predictability of the timing of such flushes, in case of a disabled **journal-datasync**, depends on the OS configuration, but without compromising (or relaxing) the process failure durability semantics as described above. Rely on the OS page cache sacrifice the power failure protection, while increasing the effectiveness of the journal operations, capable of exploiting the read caching and write combining features provided by the OS’s kernel page cache subsystem.

### Note on disabling disk write cache

Most disks contain hardware write caches. A write cache can increase the apparent performance of the disk because writes just go into the cache and are then lazily written to the disk later.

This happens irrespective of whether you have executed a fsync() from the operating system or correctly synced data from inside a Java program!

By default many systems ship with disk write cache enabled. This means that even after syncing from the operating system there is no guarantee the data has actually made it to disk, so if a failure occurs, critical data can be lost.

Some more expensive disks have non volatile or battery backed write caches which won’t necessarily lose data on event of failure, but you need to test them!
If your disk does not have an expensive non volatile or battery backed cache and it's not part of some kind of redundant array (e.g. RAID), and you value your data integrity you need to make sure disk write cache is disabled.

Be aware that disabling disk write cache can give you a nasty shock performance wise. If you've been used to using disks with write cache enabled in their default setting, unaware that your data integrity could be compromised, then disabling it will give you an idea of how fast your disk can perform when acting really reliably.

On Linux you can inspect and/or change your disk’s write cache settings using the tools `hdparm` (for IDE disks) or `sdparm` or `sginfo` (for SDSI/SATA disks).

On Windows you can check / change the setting by right clicking on the disk and clicking properties.

### 62.1.6. Installing AIO

The Java NIO journal gives great performance, but If you are running Apache ActiveMQ Artemis using Linux Kernel 2.6 or later, we highly recommend you use the ASYNCIO journal for the very best persistence performance.

It’s not possible to use the ASYNCIO journal under other operating systems or earlier versions of the Linux kernel.

If you are running Linux kernel 2.6 or later and don’t already have libaio installed, you can easily install it using the following steps:

Using yum, (e.g. on Fedora or Red Hat Enterprise Linux):

```
yum install libaio
```

Using aptitude, (e.g. on Ubuntu or Debian system):

```
apt-get install libaio
```

### 62.2. JDBC Persistence

The Apache ActiveMQ Artemis JDBC persistence layer offers the ability to store broker state (messages, address & queue definitions, etc.) using a database.

Using the ActiveMQ Artemis File Journal is the **recommended** configuration as it offers higher levels of performance and is more mature. Performance for both paging and large messages is especially diminished with JDBC. The JDBC persistence layer is targeted to those users who *must* use a database e.g. due to internal company policy.
ActiveMQ Artemis currently has support for a limited number of database vendors:

1. PostgreSQL
2. MySQL
3. Microsoft SQL Server
4. Oracle
5. DB2
6. Apache Derby

The JDBC store uses a JDBC connection to store messages and bindings data in records in database tables. The data stored in the database tables is encoded using Apache ActiveMQ Artemis internal encodings.

62.2.1. Configuring JDBC Persistence

To configure Apache ActiveMQ Artemis to use a database for persisting messages and bindings data you must do two things.

1. See the documentation on adding runtime dependencies to understand how to make the JDBC driver available to the broker.
2. Create a store element in your broker.xml config file under the <core> element. For example:

```
<store>
  <database-store>
    <jdbc-driver-class-name>org.apache.derby.jdbc.EmbeddedDriver</jdbc-driver-class-name>
    <jdbc-connection-url>jdbc:derby:databasestore;create=true</jdbc-connection-url>
    <bindings-table-name>BINDINGS_TABLE</bindings-table-name>
    <message-table-name>MESSAGE_TABLE</message-table-name>
    <page-store-table-name>MESSAGE_TABLE</page-store-table-name>
    <large-message-table-name>LARGE_MESSAGES_TABLE</large-message-table-name>
    <node-manager-store-table-name>NODE_MANAGER_TABLE</node-manager-store-table-name>
  </database-store>
</store>
```

jdbc-connection-url

The full JDBC connection URL for your database server. The connection url should include all configuration parameters and database name.

When configuring the server using the XML configuration files please ensure to escape any illegal chars; "&" for example, is typical in JDBC connection url and should be escaped to "&".
**bindings-table-name**

The name of the table in which bindings data will be persisted for the ActiveMQ Artemis server. Specifying table names allows users to share single database amongst multiple servers, without interference.

**message-table-name**

The name of the table in which bindings data will be persisted for the ActiveMQ Artemis server. Specifying table names allows users to share single database amongst multiple servers, without interference.

**large-message-table-name**

The name of the table in which messages and related data will be persisted for the ActiveMQ Artemis server. Specifying table names allows users to share single database amongst multiple servers, without interference.

**page-store-table-name**

The name of the table to house the page store directory information. Note that each address will have its own page table which will use this name appended with a unique id of up to 20 characters.

**node-manager-store-table-name**

The name of the table in which the HA Shared Store locks (i.e. primary and backup) and HA related data will be persisted for the ActiveMQ Artemis server. Specifying table names allows users to share single database amongst multiple servers, without interference. Each Shared Store primary/backup pairs must use the same table name and isn't supported to share the same table between multiple (and unrelated) primary/backup pairs.

**jdbc-driver-class-name**

The fully qualified class name of the desired database Driver.

**jdbc-network-timeout**

The JDBC network connection timeout in milliseconds. The default value is 20000 milliseconds (ie 20 seconds). When using a shared store it is recommended to set it less then or equal to `jdbc-lock-expiration`.

**jdbc-lock-renew-period**

The period in milliseconds of the keep alive service of a JDBC lock. The default value is 2000 milliseconds (ie 2 seconds).

**jdbc-lock-expiration**

The time in milliseconds a JDBC lock is considered valid without keeping it alive. The default value is 20000 milliseconds (ie 20 seconds).

**jdbc-journal-sync-period**

The time in milliseconds the journal will be synced with JDBC. The default value is 5 milliseconds.
jdbc-allowed-time-diff

The maximal time offset between the broker and the database in milliseconds when requesting the current time of the database while updating and validating primary and backup locks. Currently this value only affects the logging and will show a warning if the detected difference exceeds the limit. The default value is 250 milliseconds.

jdbc-max-page-size-bytes

The maximal size a page can use. The default and recommended maximum value is 100K bytes. Using larger sizes will result in downloading large blobs that would affect performance when using paged messages.

Some DBMS (e.g. Oracle, 30 chars) have restrictions on the size of table names, this should be taken into consideration when configuring table names for the Artemis database store, pay particular attention to the page store table name, which can be appended with a unique ID of up to 20 characters. (for Oracle this would mean configuring a page-store-table-name of max size of 10 chars).

It is also possible to explicitly add the user and password rather than in the JDBC url if you need to encode it, this would look like:

```xml
<store>
  <database-store>
    <jdbc-driver-class-name>org.apache.derby.jdbc.EmbeddedDriver</jdbc-driver-class-name>
    <jdbc-connection-url>jdbc:derby:database-store;create=true</jdbc-connection-url>
    <jdbc-user>ENC(dasfn353cewc)</jdbc-user>
    <jdbc-password>ENC(ucwiurfjtw345)</jdbc-password>
    <bindings-table-name>BINDINGS_TABLE</bindings-table-name>
    <message-table-name>MESSAGE_TABLE</message-table-name>
    <page-store-table-name>MESSAGE_TABLE</page-store-table-name>
    <large-message-table-name>LARGE_MESSAGES_TABLE</large-message-table-name>
    <node-manager-store-table-name>NODE_MANAGER_TABLE</node-manager-store-table-name>
    <jdbc-page-max-size-bytes>100K</jdbc-page-max-size-bytes>
  </database-store>
</store>
```

62.2.2. Configuring JDBC connection pooling

To configure Apache ActiveMQ Artemis to use a database with a JDBC connection pool you need to set the data source properties, for example:

```xml
<store>
  <database-store>
    <data-source-properties>
      <data-source-property key="driverClassName" value="com.mysql.jdbc.Driver"
```
You can find the documentation of the data source properties at https://commons.apache.org/proper/commons-dbcp/configuration.html.

To mask the value of a property you can use the same procedure used to mask passwords.

Please note that the reconnection works only if there is no client sending messages. Instead, if there is an attempt to write to the journal's tables during the reconnection, then the broker will fail fast and shutdown.

### 62.3. Zero Persistence

In some situations, zero persistence is sometimes required for a messaging system. Configuring Apache ActiveMQ Artemis to perform zero persistence is straightforward. Simply set the parameter `persistence-enabled` in `broker.xml` to `false`.

Please note that if you set this parameter to false, then zero persistence will occur. That means no bindings data, message data, large message data, duplicate id caches or paging data will be persisted.
Chapter 63. Data Tools

You can use the Artemis CLI to execute data maintenance tools:

The following sub-commands are available when running the CLI data command from a particular broker instance that has already been installed using the create command:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print</td>
<td>Prints a report about journal records of a non-running server</td>
</tr>
<tr>
<td>exp</td>
<td>Export the message data using a special and independent XML format</td>
</tr>
<tr>
<td>imp</td>
<td>Imports the journal to a running broker using the output from exp</td>
</tr>
<tr>
<td>encode</td>
<td>Shows an internal format of the journal encoded to String</td>
</tr>
<tr>
<td>decode</td>
<td>Imports the internal journal format from encode</td>
</tr>
<tr>
<td>compact</td>
<td>Compacts the journal of a non running server</td>
</tr>
<tr>
<td>recover</td>
<td>Recover (undelete) messages from an existing journal and create a new one.</td>
</tr>
</tbody>
</table>

You can use the CLI help for more information on how to execute each of the tools. For example:

```
$ ./artemis help data print
NAME
artemis data print - Print data records information (WARNING: don't use while a production server is running)

SYNOPSIS
artemis data print [--bindings <binding>] [--broker <brokerConfig>]
  [--f] [--jdbc] [--jdbc-bindings-table-name <jdbcBindings>]
  [--jdbc-connection-url <jdbcURL>]
  [--jdbc-driver-class-name <jdbcClassName>]
  [--jdbc-large-message-table-name <jdbcLargeMessages>]
  [--jdbc-message-table-name <jdbcMessages>]
  [--jdbc-node-manager-table-name <jdbcNodeManager>]
  [--jdbc-page-store-table-name <jdbcPageStore>] [--journal <journal>]
  [--large-messages <largeMessages>] [--output <output>]
  [<configuration>]

OPTIONS
--bindings <binding>
The folder used for bindings (default from broker.xml)

--broker <brokerConfig>
```
This would override the broker configuration from the bootstrap

--f
This will allow certain tools like print-data to be performed ignoring any running servers. WARNING: Changing data concurrently with a running broker may damage your data. Be careful with this option.

--jdbc
It will activate jdbc

--jdbc-bindings-table-name <jdbcBindings>
Name of the jdbc bindings table

--jdbc-connection-url <jdbcURL>
The connection used for the database

--jdbc-driver-class-name <jdbcClassName>
JDBC driver classname

--jdbc-large-message-table-name <jdbcLargeMessages>
Name of the large messages table

--jdbc-message-table-name <jdbcMessages>
Name of the jdbc messages table

--jdbc-node-manager-table-name <jdbcNodeManager>
Name of the jdbc node manager table

--jdbc-page-store-table-name <jdbcPageStore>
Name of the page store messages table

--journal <journal>
The folder used for messages journal (default from broker.xml)

--large-messages <largeMessages>
The folder used for large-messages (default from broker.xml)

--output <output>
Output name for the file

--paging <paging>
The folder used for paging (default from broker.xml)

--reclaimed
This option will try to print as many records as possible from reclaimed files

--safe
It will print your data structure without showing your data
--verbose
   Adds more information on the execution

--
   This option can be used to separate command-line options from the
   list of argument, (useful when arguments might be mistaken for
   command-line options

<configuration>
   Broker Configuration URI, default
   'xml:${ARTEMIS_INSTANCE}/etc/bootstrap.xml'

For a full list of data tools commands available use:

$ ./artemis help data
NAME
   artemis data - data tools group
      (print|imp|exp|encode|decode|compact|recover) (example ./artemis data
      print)
SYNOPSIS
   artemis data
   artemis data compact [--journal <journal>]
      [--large-messages <largeMessages>] [--paging <paging>]
      [--broker <brokerConfig>] [--bindings <binding>] [--verbose]
   artemis data decode [--journal <journal>]
      [--large-messages <largeMessages>] [--file-size <size>]
      [--paging <paging>] [--prefix <prefix>] [--suffix <suffix>]
      [--broker <brokerConfig>] [--directory <directory>]
      [--bindings <binding>] --verbose --input <input>
   artemis data encode [--journal <journal>]
      [--large-messages <largeMessages>] [--file-size <size>]
      [--paging <paging>] [--prefix <prefix>] [--suffix <suffix>]
      [--broker <brokerConfig>] [--directory <directory>]
   artemis data exp [--jdbc-driver-class-name <jdbcClassName>]
      [--journal <journal>] [--jdbc-connection-url <jdbcURL>]
      [--large-messages <largeMessages>]
      [--jdbc-bindings-table-name <jdbcBindings>] [--paging <paging>] [--f]
      [--jdbc-large-message-table-name <jdbcLargeMessages>]
      [--broker <brokerConfig>] [--jdbc-page-store-table-name
      <jdbcPageStore>]
      [--bindings <binding>] --jdbc --verbose
      --jdbc-message-table-name <jdbcMessages>
      --jdbc-node-manager-table-name <jdbcNodeManager>
   artemis data imp [--legacy-prefixes] [--password <password>]
      [--jdbc-driver-class-name <jdbcClassName>]
      [--jdbc-connection-url <jdbcURL>]
      [--large-messages <largeMessages>]
      [--jdbc-bindings-table-name <jdbcBindings>] [--output <output>]
      [--jdbc-node-manager-table-name <jdbcNodeManager>]
   artemis data print [-reclaimed]
      [-jdbc-driver-class-name <jdbcClassName>] [-journal <journal>]
With no arguments, Display help information

recover
Recover (undelete) every message on the journal by creating a new output journal. Rolled backed and acked messages will be sent out to the output as much as possible.

With --jdbc-driver-class-name option, JDBC driver classname

With --journal option, The folder used for messages journal (default from broker.xml)

With --jdbc-connection-url option, The connection used for the database

With --large-messages option, The folder used for large-messages (default from broker.xml)

With --reclaimed option, This option will try to recover as many records as possible from reclaimed files

With --target option, Output folder container the new journal with all the generated messages

With --jdbc-bindings-table-name option, Name of the jdbc bindigns table

With --paging option, The folder used for paging (default from broker.xml)
With --f option, This will allow certain tools like print-data to be performed ignoring any running servers. WARNING: Changing data concurrently with a running broker may damage your data. Be careful with this option.

With --jdbc-large-message-table-name option, Name of the large messages table

With --broker option, This would override the broker configuration from the bootstrap

With --jdbc-page-store-table-name option, Name of the page store messages table

With --bindings option, The folder used for bindings (default from broker.xml)

With --jdbc option, It will activate jdbc

With --verbose option, Adds more information on the execution

With --jdbc-message-table-name option, Name of the jdbc messages table

With --jdbc-node-manager-table-name option, Name of the jdbc node manager table

With --output option, Output name for the file

print
Print data records information (WARNING: don't use while a production server is running)

With --reclaimed option, This option will try to print as many records as possible from reclaimed files

With --jdbc-driver-class-name option, JDBC driver classname

With --journal option, The folder used for messages journal (default from broker.xml)

With --jdbc-connection-url option, The connection used for the database

With --large-messages option, The folder used for large-messages (default from broker.xml)

With --jdbc-bindings-table-name option, Name of the jdbc bindings table

With --paging option, The folder used for paging (default from
With `--f` option, this will allow certain tools like `print-data` to be performed ignoring any running servers. WARNING: Changing data concurrently with a running broker may damage your data. Be careful with this option.

With `--jdbc-large-message-table-name` option, Name of the large messages table

With `--safe` option, it will print your data structure without showing your data

With `--broker` option, this would override the broker configuration from the bootstrap

With `--jdbc-page-store-table-name` option, Name of the page store messages table

With `--bindings` option, The folder used for bindings (default from broker.xml)

With `--jdbc` option, it will activate jdbc

With `--verbose` option, adds more information on the execution

With `--jdbc-message-table-name` option, Name of the jdbc messages table

With `--jdbc-node-manager-table-name` option, Name of the jdbc node manager table

With `--output` option, Output name for the file

`exp` Export all message-data using an XML that could be interpreted by any system.

With `--jdbc-driver-class-name` option, JDBC driver classname

With `--journal` option, The folder used for messages journal (default from broker.xml)

With `--jdbc-connection-url` option, The connection used for the database

With `--large-messages` option, The folder used for large-messages (default from broker.xml)

With `--jdbc-bindings-table-name` option, Name of the jdbc bindings table
With --paging option, The folder used for paging (default from broker.xml)

With --f option, This will allow certain tools like print-data to be performed ignoring any running servers. WARNING: Changing data concurrently with a running broker may damage your data. Be careful with this option.

With --jdbc-large-message-table-name option, Name of the large messages table

With --broker option, This would override the broker configuration from the bootstrap

With --jdbc-page-store-table-name option, Name of the page store messages table

With --bindings option, The folder used for bindings (default from broker.xml)

With --jdbc option, It will activate jdbc

With --verbose option, Adds more information on the execution

With --jdbc-message-table-name option, Name of the jdbc messages table

With --jdbc-node-manager-table-name option, Name of the jdbc node manager table

With --output option, Output name for the file

imp
Import all message-data using an XML that could be interpreted by any system.

With --legacy-prefixes option, Do not remove prefixes from legacy imports

With --password option, User name used to import the data. (default null)

With --transaction option, If this is set to true you will need a whole transaction to commit at the end. (default false)

With --verbose option, Adds more information on the execution

With --port option, The port used to import the data (default 61616)

With --user option, User name used to import the data. (default
null)

With --sort option, Sort the messages from the input (used for older versions that won't sort messages)

With --input option, The input file name (default=exp.dmp)

With --host option, The host used to import the data (default localhost)

decode
Decode a journal's internal format into a new journal set of files

With --journal option, The folder used for messages journal (default from broker.xml)

With --large-messages option, The folder used for large-messages (default from broker.xml)

With --file-size option, The journal size (default 10485760)

With --paging option, The folder used for paging (default from broker.xml)

With --prefix option, The journal prefix (default activemq-data)

With --suffix option, The journal suffix (default amq)

With --broker option, This would override the broker configuration from the bootstrap

With --directory option, The journal folder (default journal folder from broker.xml)

With --bindings option, The folder used for bindings (default from broker.xml)

With --verbose option, Adds more information on the execution

With --input option, The input file name (default=exp.dmp)

encode
Encode a set of journal files into an internal encoded data format

With --journal option, The folder used for messages journal (default from broker.xml)

With --large-messages option, The folder used for large-messages (default from broker.xml)

With --file-size option, The journal size (default 10485760)
With --paging option, The folder used for paging (default from broker.xml)

With --prefix option, The journal prefix (default activemq-data)

With --suffix option, The journal suffix (default amq)

With --broker option, This would override the broker configuration from the bootstrap

With --bindings option, The folder used for bindings (default from broker.xml)

With --verbose option, Adds more information on the execution

With --directory option, The journal folder (default the journal folder from broker.xml)

compact
  Compacts the journal of a non running server

With --journal option, The folder used for messages journal (default from broker.xml)

With --large-messages option, The folder used for large-messages (default from broker.xml)

With --paging option, The folder used for paging (default from broker.xml)

With --broker option, This would override the broker configuration from the bootstrap

With --bindings option, The folder used for bindings (default from broker.xml)

With --verbose option, Adds more information on the execution
Chapter 64. Libaio Native Libraries

Apache ActiveMQ Artemis distributes a native library, used as a bridge for its fast journal, between Apache ActiveMQ Artemis and Linux libaio.

Libaio is a library, developed as part of the Linux kernel project. With libaio we submit writes to the operating system where they are processed asynchronously. Some time later the OS will call our code back when they have been processed.

We use this in our high performance journal if configured to do so, please see Persistence.

These are the native libraries distributed by Apache ActiveMQ Artemis:

- libartemis-native-64.so - x86 64 bits
- We distributed a 32-bit version until early 2017. While it’s not available on the distribution any longer it should still be possible to compile to a 32-bit environment if needed.

When using libaio, Apache ActiveMQ Artemis will always try loading these files as long as they are on the library path.

### 64.1. Runtime dependencies

If you just want to use the provided native binaries you need to install the required libaio dependency.

You can install libaio using the following steps as the root user:

Using yum, (e.g. on Fedora or Red Hat Enterprise Linux):

```
yum install libaio
```

Using aptitude, (e.g. on Ubuntu or Debian system):

```
apt-get install libaio1
```

### 64.2. Compiling the native libraries

In the case that you are using Linux on a platform other than x86_32 or x86_64 (for example Itanium 64 bits or IBM Power) you may need to compile the native library, since we do not distribute binaries for those platforms with the release.

### 64.3. Compilation dependencies

The native layer is only available on Linux. If you are in a platform other than Linux the native compilation will not work.
These are the required Linux packages to be installed for the compilation to work:

- gcc - C Compiler
- gcc-c++ or g++ - Extension to gcc with support for C++
- libtool - Tool for link editing native libraries
- libaio - Library to disk asynchronous IO kernel functions
- libaio-dev - Compilation support for libaio
- cmake
- A full JDK installed with the environment variable JAVA_HOME set to its location

To perform this installation on RHEL or Fedora, you can simply type this at a command line:

```
sudo yum install libtool gcc-c++ gcc libaio libaio-devel cmake
```

Or on Debian systems:

```
sudo apt-get install libtool gcc-g++ gcc libaio libaio-dev cmake
```

You could find a slight variation of the package names depending on the version and Linux distribution. (for example gcc-c++ on Fedora versus g++ on Debian systems)

### 64.4. Invoking the compilation

In the source distribution or git clone, in the `artemis-native` directory, execute the shell script `compile-native.sh`. This script will invoke the proper commands to perform the native build.

If you want more information refer to the [cmake web pages](https://cmake.org).
Chapter 65. Detecting Dead Connections

In this section we will discuss connection time-to-live (TTL) and explain how Apache ActiveMQ Artemis deals with crashed clients and clients which have exited without cleanly closing their resources.

65.1. Cleaning up Resources on the Server

Before an Apache ActiveMQ Artemis client application exits it is considered good practice that it should close its resources in a controlled manner, using a `finally` block.

Here's an example of a well behaved core client application closing its session and session factory in a finally block:

```java
ServerLocator locator = null;
ClientSessionFactory sf = null;
ClientSession session = null;

try {
    locator = ActiveMQClient.createServerLocatorWithoutHA(...);
    sf = locator.createSessionFactory();
    session = sf.createSession(...);

    ... do some stuff with the session...
} finally {
    if (session != null) {
        session.close();
    }

    if (sf != null) {
        sf.close();
    }

    if(locator != null) {
        locator.close();
    }
}
```

And here's an example of a well behaved JMS client application:

```java
Connection jmsConnection = null;

try {
   ConnectionFactory jmsConnectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");
```
```java
jmsConnection = jmsConnectionFactory.createConnection();

... do some stuff with the connection...
}
finally {
    if (connection != null) {
        connection.close();
    }
}
```

Or with using auto-closeable feature from Java, which can save a few lines of code:

```java
try {
    ActiveMQConnectionFactory jmsConnectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");
    Connection jmsConnection = jmsConnectionFactory.createConnection();
    ... do some stuff with the connection...
}
```

Unfortunately users don’t always write well behaved applications, and sometimes clients just crash so they don’t have a chance to clean up their resources!

If this occurs then it can leave server side resources, like sessions, hanging on the server. If these were not removed they would cause a resource leak on the server and over time this result in the server running out of memory or other resources.

We have to balance the requirement for cleaning up dead client resources with the fact that sometimes the network between the client and the server can fail and then come back, allowing the client to reconnect. Apache ActiveMQ Artemis supports client reconnection, so we don’t want to clean up “dead” server side resources too soon or this will prevent any client from reconnecting, as it won’t be able to find its old sessions on the server.

Apache ActiveMQ Artemis makes all of this configurable via a connection TTL. Basically, the TTL determines how long the server will keep a connection alive in the absence of any data arriving from the client. The client will automatically send “ping” packets periodically to prevent the server from closing it down. If the server doesn’t receive any packets on a connection for the connection TTL time, then it will automatically close all the sessions on the server that relate to that connection.

The connection TTL is configured on the URI using the connectionTTL parameter.

The default value for connection ttl on an “unreliable” connection (e.g. a Netty connection using the tcp URL scheme) is 60000ms, i.e. 1 minute. The default value for connection ttl on a “reliable” connection (e.g. an in-vm connection using the vm URL scheme) is -1. A value of -1 for connectionTTL means the server will never time out the connection on the server side.

If you do not wish clients to be able to specify their own connection TTL, you can override all values used by a global value set on the server side. This can be done by specifying the connection-ttl-override attribute in the server side configuration. The default value for connection-ttl-override is -1 which means "do not override" (i.e. let clients use their own values).
The logic to check connections for TTL violations runs periodically on the broker. By default, the checks are done every 2,000 milliseconds. However, this can be changed if necessary by using the `connection-ttl-check-interval` attribute.

### 65.2. Closing Forgotten Resources

As previously discussed, it's important that all core client sessions and JMS connections are always closed explicitly in a `finally` block when you are finished using them.

If you fail to do so, Apache ActiveMQ Artemis will detect this at garbage collection time, and log a warning (If you are using JMS the warning will involve a JMS connection).

Apache ActiveMQ Artemis will then close the connection / client session for you.

Note that the log will also tell you the exact line of your user code where you created the JMS connection / client session that you later did not close. This will enable you to pinpoint the error in your code and correct it appropriately.

### 65.3. Detecting Failure from the Client

In the previous section we discussed how the client sends pings to the server and how "dead" connection resources are cleaned up by the server. There's also another reason for pinging, and that's for the client to be able to detect that the server or network has failed.

As long as the client is receiving data from the server it will consider the connection to be still alive.

If the client does not receive any packets for a configurable number of milliseconds then it will consider the connection failed and will either initiate failover, or call any `FailureListener` instances (or `ExceptionListener` instances if you are using JMS) depending on how it has been configured.

This is controlled by setting the `clientFailureCheckPeriod` parameter on the URI your client is using to connect, e.g. `tcp://localhost:61616?clientFailureCheckPeriod=30000`.

The default value for client failure check period on an "unreliable" connection (e.g. a Netty connection) is 30000 ms, i.e. 30 seconds. The default value for client failure check period on a "reliable" connection (e.g. an in-vm connection) is -1. A value of -1 means the client will never fail the connection on the client side if no data is received from the server. Typically this is much lower than connection TTL to allow clients to reconnect in case of transitory failure.

### 65.4. Configuring Asynchronous Connection Execution

Most packets received on the server side are executed on the remoting thread. These packets represent short-running operations and are always executed on the remoting thread for performance reasons.

However, by default some kinds of packets are executed using a thread from a thread pool so that the remoting thread is not tied up for too long. Please note that processing operations asynchronously on another thread adds a little more latency. These packets are:
• org.apache.activemq.artemis.core.protocol.core.impl.wireformat.RollbackMessage
• org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionCloseMessage
• org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionCommitMessage
• org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionXACommitMessage
• org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionXAPrepareMessage
• org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionXARollbackMessage

To disable asynchronous connection execution, set the parameter async-connection-execution-enabled in broker.xml to false (default value is true).
Chapter 66. Configuring the Transport

In this chapter we’ll describe the concepts required for understanding Apache ActiveMQ Artemis transports and where and how they’re configured.

66.1. Acceptors

One of the most important concepts in Apache ActiveMQ Artemis transports is the acceptor. Let’s dive straight in and take a look at an acceptor defined in xml in the configuration file broker.xml.

```xml
<acceptor name="netty">tcp://localhost:61617</acceptor>
```

Acceptors are always defined inside an acceptors element. There can be one or more acceptors defined in the acceptors element. There’s no upper limit to the number of acceptors per server.

Each acceptor defines a way in which connections can be made to the Apache ActiveMQ Artemis server.

In the above example we’re defining an acceptor that uses Netty to listen for connections at port 61617.

The acceptor element contains a URL that defines the kind of Acceptor to create along with its configuration. The schema part of the URL defines the Acceptor type which can either be tcp or vm which is Netty or an In VM Acceptor respectively. For Netty the host and the port of the URL define what host and port the acceptor will bind to. For In VM the Authority part of the URL defines a unique server id.

The acceptor can also be configured with a set of key=value pairs used to configure the specific transport, the set of valid key=value pairs depends on the specific transport be used and are passed straight through to the underlying transport. These are set on the URL as part of the query, like so:

```xml
<acceptor name="netty">
 tcp://localhost:61617?sslEnabled=true;keyStorePath=/path</acceptor>
```

66.2. Connectors

Whereas acceptors are used on the server to define how we accept connections, connectors are used to define how to connect to a server.

Let’s look at a connector defined in our broker.xml file:

```xml
<connector name="netty">tcp://localhost:61617</connector>
```

Connectors can be defined inside a connectors element. There can be one or more connectors defined in the connectors element. There’s no upper limit to the number of connectors per server.
A connector is used when the server acts as a client itself, e.g.:

- When one server is bridged to another
- When a server takes part in a cluster

In these cases the server needs to know how to connect to other servers. That's defined by connectors.

### 66.3. Configuring the Transport Directly from the Client

How do we configure a core `ClientSessionFactory` with the information that it needs to connect with a server?

Connectors are also used indirectly when configuring a core `ClientSessionFactory` to directly talk to a server. Although in this case there's no need to define such a connector in the server side configuration, instead we just specify the appropriate URI.

Here's an example of creating a `ClientSessionFactory` which will connect directly to the acceptor we defined earlier in this chapter, it uses the standard Netty TCP transport and will try and connect on port 61617 to localhost (default):

```java
ServerLocator locator = ActiveMQClient.createServerLocator("tcp://localhost:61617");

ClientSessionFactory sessionFactory = locator.createSessionFactory();

ClientSession session = sessionFactory.createSession(...);
```

Similarly, if you're using JMS, you can configure the JMS connection factory directly on the client side:

```java
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61617");

Connection jmsConnection = connectionFactory.createConnection();
```

### 66.4. Configuring the Netty transport

Out of the box, Apache ActiveMQ Artemis currently uses Netty, a high performance low level network library.

Our Netty transport can be configured in several different ways; to use straightforward TCP sockets, SSL, or to tunnel over HTTP or HTTPS.

We believe this caters for the vast majority of transport requirements.
66.4.1. Single Port Support

Apache ActiveMQ Artemis supports using a single port for all protocols, Apache ActiveMQ Artemis will automatically detect which protocol is being used CORE, AMQP, STOMP, MQTT or OPENWIRE and use the appropriate Apache ActiveMQ Artemis handler. It will also detect whether protocols such as HTTP or Web Sockets are being used and also use the appropriate decoders. Web Sockets are supported for AMQP, STOMP, and MQTT.

It is possible to limit which protocols are supported by using the protocols parameter on the Acceptor like so:

```xml
<acceptor name="netty">tcp://localhost:61617?protocols=CORE,AMQP</acceptor>
```

66.4.2. Configuring Netty TCP

Netty TCP is a simple unencrypted TCP sockets based transport. If you're running connections across an untrusted network please bear in mind this transport is unencrypted. You may want to look at the SSL or HTTPS configurations.

With the Netty TCP transport all connections are initiated from the client side (i.e. the server does not initiate any connections to the client). This works well with firewall policies that typically only allow connections to be initiated in one direction.

All the valid keys for the tcp URL scheme used for Netty are defined in the class org.apache.activemq.artemis.core.remoting.impl.netty.TransportConstants. Most parameters can be used either with acceptors or connectors, some only work with acceptors. The following parameters can be used to configure Netty for simple TCP:

- **The host and port parameters** are only used in the core API, in XML configuration these are set in the URI host and port.

**host**

This specifies the host name or IP address to connect to (when configuring a connector) or to listen on (when configuring an acceptor). The default value for this property is localhost. When configuring acceptors, multiple hosts or IP addresses can be specified by separating them with commas. It is also possible to specify 0.0.0.0 to accept connections from all the host’s network interfaces. It’s not valid to specify multiple addresses when specifying the host for a connector; a connector makes a connection to one specific address.

Don’t forget to specify a host name or IP address! If you want your server able to accept connections from other nodes you must specify a hostname or IP address at which the acceptor will bind and listen for incoming connections. The default is localhost which of course is not accessible from remote nodes!

**port**

This specified the port to connect to (when configuring a connector) or to listen on (when configuring an acceptor). The default value for this property is 61616.
tcpNoDelay

If this is true then Nagle’s algorithm will be disabled. This is a Java (client) socket option. The default value for this property is true.

tcpSendBufferSize

This parameter determines the size of the TCP send buffer in bytes. The default value for this property is 32768 bytes (32KiB).

TCP buffer sizes should be tuned according to the bandwidth and latency of your network. Here’s a good link that explains the theory behind this.

In summary TCP send/receive buffer sizes should be calculated as:

\[ \text{buffer size} = \text{bandwidth} \times \text{RTT}. \]

Where bandwidth is in bytes per second and network round trip time (RTT) is in seconds. RTT can be easily measured using the ping utility.

For fast networks you may want to increase the buffer sizes from the defaults.

tcpReceiveBufferSize

This parameter determines the size of the TCP receive buffer in bytes. The default value for this property is 32768 bytes (32KiB).

writeBufferLowWaterMark

This parameter determines the low water mark of the Netty write buffer. Once the number of bytes queued in the write buffer exceeded the high water mark and then dropped down below this value, Netty’s channel will start to be writable again. The default value for this property is 32768 bytes (32KiB).

writeBufferHighWaterMark

This parameter determines the high water mark of the Netty write buffer. If the number of bytes queued in the write buffer exceeds this value, Netty’s channel will start to be not writable. The default value for this property is 131072 bytes (128KiB).

batchDelay

Before writing packets to the transport, Apache ActiveMQ Artemis can be configured to batch up writes for a maximum of batchDelay milliseconds. This can increase overall throughput for very small messages. It does so at the expense of an increase in average latency for message transfer. The default value for this property is 0 ms.

directDeliver

When a message arrives on the server and is delivered to waiting consumers, by default, the delivery is done on the same thread as that on which the message arrived. This gives good latency in environments with relatively small messages and a small number of consumers, but at the cost of overall throughput and scalability - especially on multi-core machines. If you want the lowest latency and a possible reduction in throughput then you can use the default value for directDeliver (i.e. true). If you are willing to take some small extra hit on latency but want the
highest throughput set directDeliver to false.

nioRemotingThreads
This is deprecated. It is replaced by remotingThreads, if you are using this please update your configuration.

remotingThreads
Apache ActiveMQ Artemis will, by default, use a number of threads equal to three times the number of cores (or hyper-threads) as reported by Runtime.getRuntime().availableProcessors() for processing incoming packets. If you want to override this value, you can set the number of threads by specifying this parameter. The default value for this parameter is -1 which means use the value from Runtime.getRuntime().availableProcessors() * 3.

localAddress
When configured a Netty Connector it is possible to specify which local address the client will use when connecting to the remote address. This is typically used in the Application Server or when running Embedded to control which address is used for outbound connections. If the local-address is not set then the connector will use any local address available.

localPort
When configured a Netty Connector it is possible to specify which local port the client will use when connecting to the remote address. This is typically used in the Application Server or when running Embedded to control which port is used for outbound connections. If the local-port default is used, which is 0, then the connector will let the system pick up an ephemeral port. Valid ports are 0 to 65535.

connectionsAllowed
This is only valid for acceptors. It limits the number of connections which the acceptor will allow. When this limit is reached a DEBUG level message is issued to the log, and the connection is refused. The type of client in use will determine what happens when the connection is refused. In the case of a core client, it will result in a org.apache.activemq.artemis.api.core.ActiveMQConnectionTimedOutException. Default value is -1 (unlimited).

handshake-timeout
Prevents an unauthorised client opening a large number of connections and just keeping them open. As connections each require a file handle this consumes resources that are then unavailable to other clients. Once the connection is authenticated, the usual rules can be enforced regarding resource consumption. Default value is set to 10 seconds. Each integer is valid value. When set value to zero or negative integer this feature is turned off. Changing value needs to restart server to take effect.

autoStart
Determines whether or not an acceptor will start automatically when the broker is started. Default value is true.
66.4.3. Configuring Netty Native Transport

Netty Native Transport support exists for selected OS platforms. This allows Apache ActiveMQ Artemis to use native sockets/io instead of Java NIO.

These Native transports add features specific to a particular platform, generate less garbage, and generally improve performance when compared to Java NIO based transport.

Both Clients and Server can benefit from this.

Current Supported Platforms.

- Linux running 64bit JVM
- MacOS running 64bit JVM

Apache ActiveMQ Artemis will by default enable the corresponding native transport if a supported platform is detected.

If running on an unsupported platform or any issues loading native libs, Apache ActiveMQ Artemis will fallback onto Java NIO.

Linux Native Transport

On supported Linux platforms Epoll is used, @see https://en.wikipedia.org/wiki/Epoll.

The following properties are specific to this native transport:

useEpoll

  enables the use of epoll if a supported linux platform is running a 64bit JVM is detected. Setting this to false will force the use of Java NIO instead of epoll. Default is true

MacOS Native Transport

On supported MacOS platforms KQueue is used, @see https://en.wikipedia.org/wiki/Kqueue.

The following properties are specific to this native transport:

useKQueue

  enables the use of kqueue if a supported MacOS platform running a 64bit JVM is detected. Setting this to false will force the use of Java NIO instead of kqueue. Default is true

66.4.4. Configuring Netty SSL

Netty SSL is similar to the Netty TCP transport but it provides additional security by encrypting TCP connections using the Secure Sockets Layer SSL.

Please see the examples for a full working example of using Netty SSL.

Netty SSL uses all the same properties as Netty TCP but adds the following additional properties:
sslContext

An optional cache key only evaluated if org.apache.activemq.artemis.core.remoting.impl.ssl.CachingSSLContextFactory is active, to cache the initial created SSL context and reuse it. If not specified CachingSSLContextFactory will automatically calculate a cache key based on the given keystore/truststore parameters. See Configuring an SSLContextFactory for more details.

sslEnabled

Must be true to enable SSL. Default is false.

sslAutoReload

Must be true to have the broker ‘watch’ an acceptors keyStorePath and/or trustStorePath and invoke reload() on update. The watch period is controlled by the configuration reload feature. Default is false.

keyStorePath

When used on an acceptor this is the path to the SSL key store on the server which holds the server's certificates (whether self-signed or signed by an authority).

When used on a connector this is the path to the client-side SSL key store which holds the client certificates. This is only relevant for a connector if you are using 2-way SSL (i.e. mutual authentication). Although this value is configured on the server, it is downloaded and used by the client. If the client needs to use a different path from that set on the server then it can override the server-side setting by either using the customary "javax.net.ssl.keyStore" system property or the ActiveMQ-specific "org.apache.activemq.ssl.keyStore" system property. The ActiveMQ-specific system property is useful if another component on the client is already making use of the standard Java system property.

keyStorePassword

When used on an acceptor this is the password for the server-side keystore.

When used on a connector this is the password for the client-side keystore. This is only relevant for a connector if you are using 2-way SSL (i.e. mutual authentication). Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different password from that set on the server then it can override the server-side setting by either using the customary "javax.net.ssl.keyStorePassword" system property or the ActiveMQ-specific "org.apache.activemq.ssl.keyStorePassword" system property. The ActiveMQ-specific system property is useful if another component on the client is already making use of the standard Java system property.

keyStoreType

The type of keystore being used. For example, JKS, JCEKS, PKCS12, PEM etc. This value can also be set via the "javax.net.ssl.keyStoreType" system property or the ActiveMQ-specific "org.apache.activemq.ssl.keyStoreType" system property. The ActiveMQ-specific system property is useful if another component on the client is already making use of the standard Java system property. Default is JKS.
keyStoreProvider

The provider used for the keystore. For example, SUN, SunJCE, etc. This value can also be set via the "javax.net.ssl.keyStoreProvider" system property or the ActiveMQ-specific "org.apache.activemq.ssl.keyStoreProvider" system property. The ActiveMQ-specific system property is useful if another component on the client is already making use of the standard Java system property. Default is null.

keyStoreAlias

When used on an acceptor this is the alias to select from the SSL key store (specified via keyStorePath) to present to the client when it connects.

When used on a connector this is the alias to select from the SSL key store (specified via keyStorePath) to present to the server when the client connects to it. This is only relevant for a connector when using 2-way SSL (i.e. mutual authentication).

Default is null.

trustStorePath

When used on an acceptor this is the path to the server-side SSL key store that holds the keys of all the clients that the server trusts. This is only relevant for an acceptor if you are using 2-way SSL (i.e. mutual authentication).

When used on a connector this is the path to the client-side SSL key store which holds the public keys of all the servers that the client trusts. Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different path from that set on the server then it can override the server-side setting by either using the customary "javax.net.ssl.trustStore" system property or the ActiveMQ-specific "org.apache.activemq.ssl.trustStore" system property. The ActiveMQ-specific system property is useful if another component on the client is already making use of the standard Java system property.

trustStorePassword

When used on an acceptor this is the password for the server-side trust store. This is only relevant for an acceptor if you are using 2-way SSL (i.e. mutual authentication).

When used on a connector this is the password for the client-side truststore. Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different password from that set on the server then it can override the server-side setting by either using the customary "javax.net.ssl.trustStorePassword" system property or the ActiveMQ-specific "org.apache.activemq.ssl.trustStorePassword" system property. The ActiveMQ-specific system property is useful if another component on the client is already making use of the standard Java system property.

trustStoreType

The type of truststore being used. For example, JKS, JCEKS, PKCS12, PEM etc. This value can also be set via the "javax.net.ssl.trustStoreType" system property or the ActiveMQ-specific "org.apache.activemq.ssl.trustStoreType" system property. The ActiveMQ-specific system property is useful if another component on the client is already making use of the standard Java system property. Default is JKS.
trustStoreProvider

The provider used for the truststore. For example, SUN, SunJCE, etc. This value can also be set via the "javax.net.ssl.trustStoreProvider" system property or the ActiveMQ-specific "org.apache.activemq.ssl.trustStoreProvider" system property. The ActiveMQ-specific system property is useful if another component on the client is already making use of the standard Java system property. Default is null.

enabledCipherSuites

Whether used on an acceptor or connector this is a comma separated list of cipher suites used for SSL communication. The default value is null which means the JVM’s default will be used.

enabledProtocols

Whether used on an acceptor or connector this is a comma separated list of protocols used for SSL communication. The default value is null which means the JVM’s default will be used.

needClientAuth

This property is only for an acceptor. It tells a client connecting to this acceptor that 2-way SSL is required. Valid values are true or false. Default is false.

This property takes precedence over wantClientAuth and if its value is set to true then wantClientAuth will be ignored.

wantClientAuth

This property is only for an acceptor. It tells a client connecting to this acceptor that 2-way SSL is requested but not required. Valid values are true or false. Default is false.

If the property needClientAuth is set to true then that property will take precedence and this property will be ignored.

verifyHost

When used on a connector the CN or Subject Alternative Name values of the server's SSL certificate will be compared with the hostname being connected to in order to verify a match. This is useful for both 1-way and 2-way SSL.

When used on an acceptor the CN or Subject Alternative Name values of the connecting client’s SSL certificate will be compared to its hostname to verify a match. This is useful only for 2-way SSL.

Valid values are true or false. Default is true for connectors, and false for acceptors.

trustAll

When used on a connector the client will trust the provided server certificate implicitly, regardless of any configured trust store.

This setting is primarily for testing purposes only and should not be used in production.

Valid values are true or false. Default is false.
**forceSSLParameters**

When used on a connector any SSL settings that are set as parameters on the connector will be used instead of JVM system properties including both javax.net.ssl and ActiveMQ system properties to configure the SSL context for this connector.

Valid values are **true** or **false**. Default is **false**.

**useDefaultSslContext**

Only valid on a connector. Allows the connector to use the "default" SSL context (via SSLContext.getDefault()) which can be set programatically by the client (via SSLContext.setDefault(SSLContext)). If set to **true** all other SSL related parameters except for sslEnabled are ignored.

Valid values are **true** or **false**. Default is **false**.

**sslProvider**

Used to change the SSL Provider between JDK and OPENSSL. The default is JDK. If used with OPENSSL you can add netty-tcnative to your classpath to use the native installed openssl. This can be useful if you want to use special ciphersuite - elliptic curve combinations which are support through openssl but not through the JDK provider. See [https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations](https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations) for more information’s.

**sniHost**

When used on an acceptor the sniHost is a regular expression used to match the server_name extension on incoming SSL connections. If the name doesn't match then the connection to the acceptor will be rejected. A WARN message will be logged if this happens. If the incoming connection doesn't include the server_name extension then the connection will be accepted.

When used on a connector the sniHost value is used for the server_name extension on the SSL connection.

**trustManagerFactoryPlugin**

This is valid on either an acceptor or connector. It defines the name of the class which implements org.apache.activemq.artemis.api.core.TrustManagerFactoryPlugin. This is a simple interface with a single method which returns a javax.net.ssl.TrustManagerFactory. The TrustManagerFactory will be used when the underlying javax.net.ssl.SSLContext is initialized. This allows fine-grained customization of who/what the broker & client trusts.

This value takes precedence of all other SSL parameters which apply to the trust manager (i.e. trustAll, truststoreProvider, truststorePath, truststorePassword, crlPath).

Any plugin specified will need to be placed on the broker's classpath.

**Configuring an SSLContextFactory**

If you use JDK as SSL provider (the default), you can configure which SSLContextFactory to use. Currently the following two implementations are provided:

- org.apache.activemq.artemis.core.remoting.impl.ssl.DefaultSSLContextFactory (registered by
the default)

- org.apache.activemq.artemis.core.remoting.impl.ssl.CachingSSLContextFactory

You may also create your own implementation of org.apache.activemq.artemis.spi.core.remoting.ssl.SSLContextFactory.

The implementations are loaded by a java.util.ServiceLoader, thus you need to declare your implementation in a META-INF/services/org.apache.activemq.artemis.spi.core.remoting.ssl.SSLContextFactory file. If several implementations are available, the one with the highest priority will be selected.

So for example, if you want to use org.apache.activemq.artemis.core.remoting.impl.ssl.CachingSSLContextFactory you need to add a META-INF/services/org.apache.activemq.artemis.spi.core.remoting.ssl.SSLContextFactory file to your classpath with the content org.apache.activemq.artemis.core.remoting.impl.ssl.CachingSSLContextFactory.

A similar mechanism exists for the OPENSSL SSL provider in which case you can configure an OpenSSLContextFactory. Currently the following two implementations are provided:

- org.apache.activemq.artemis.core.remoting.impl.ssl.DefaultOpenSSLContextFactory (registered by the default)
- org.apache.activemq.artemis.core.remoting.impl.ssl.CachingOpenSSLContextFactory

You may also create your own implementation of org.apache.activemq.artemis.spi.core.remoting.ssl.OpenSSLContextFactory.

### 66.4.5. Configuring Netty HTTP

Netty HTTP tunnels packets over the HTTP protocol. It can be useful in scenarios where firewalls only allow HTTP traffic to pass.

Please see the examples for a full working example of using Netty HTTP.

Netty HTTP uses the same properties as Netty TCP but adds the following additional properties:

- **httpEnabled**
  Activates http on the client. This is not needed on the broker. With single port support Apache ActiveMQ Artemis will now automatically detect if http is being used and configure itself.

- **httpClientIdleTime**
  How long a client can be idle before sending an empty http request to keep the connection alive

- **httpClientIdleScanPeriod**
  How often, in milliseconds, to scan for idle clients

- **httpResponseTime**
  How long the server can wait before sending an empty http response to keep the connection alive
httpServerScanPeriod
How often, in milliseconds, to scan for clients needing responses

httpRequiresSessionId
If true the client will wait after the first call to receive a session id. Used the http connector is connecting to servlet acceptor (not recommended)

66.4.6. Configuring Netty SOCKS Proxy

All these parameters are only applicable to a connector and/or client URL.

Using a loop-back address (e.g. localhost or 127.0.0.1) as the target of the connector or URL will circumvent the application of these configuration properties. In other words, no SOCKS proxy support will be configured even if these properties are set.

socksEnabled
Whether or not to enable SOCKS support on the connector.

socksHost
The name of the SOCKS server to use.

socksPort
The port of the SOCKS server to use.

socksVersion
The version of SOCKS to use. Must be an integer. Default is 5.

socksUsername
The username to use when connecting to the socksHost.

socksPassword
The password to use when connecting to the socksHost. Only applicable if the socksVersion is 5.

socksRemoteDNS
Whether or not to create remote destination socket unresolved and disable DNS resolution. Default is false.
Chapter 67. Flow Control

Flow control is used to limit the flow of data between a client and server, or a server and another server in order to prevent the client or server being overwhelmed with data.

67.1. Consumer Flow Control

This controls the flow of data between the server and the client as the client consumes messages. For performance reasons clients normally buffer messages before delivering to the consumer via the `receive()` method or asynchronously via a message listener. If the consumer cannot process messages as fast as they are being delivered and stored in the internal buffer, then you could end up with a situation where messages would keep building up possibly causing out of memory on the client if they cannot be processed in time.

67.2. Window-Based Flow Control

By default, Apache ActiveMQ Artemis consumers buffer messages from the server in a client side buffer before the client consumes them. This improves performance: otherwise every time the client consumes a message, Apache ActiveMQ Artemis would have to go the server to request the next message. In turn, this message would then get sent to the client side, if one was available.

A network round trip would be involved for every message and considerably reduce performance.

To prevent this, Apache ActiveMQ Artemis pre-fetches messages into a buffer on each consumer. The total maximum size of messages (in bytes) that will be buffered on each consumer is determined by the `consumerWindowSize` parameter.

By default, the `consumerWindowSize` is set to 1 MiB (1024 * 1024 bytes) unless overridden via (Address Settings)

The value can be:

- -1 for an unbounded buffer
- 0 to not buffer any messages.
- >0 for a buffer with the given maximum size in bytes.

Setting the consumer window size can considerably improve performance depending on the messaging use case. As an example, let's consider the two extremes:

67.2.1. Fast consumers

Fast consumers can process messages as fast as they consume them (or even faster)

To allow fast consumers, set the `consumerWindowSize` to -1. This will allow unbounded message buffering on the client side.

Use this setting with caution: it can overflow the client memory if the consumer is not able to process messages as fast as it receives them.
67.2.2. Slow consumers

Slow consumers takes significant time to process each message and it is desirable to prevent buffering messages on the client side so that they can be delivered to another consumer instead.

Consider a situation where a queue has 2 consumers; 1 of which is very slow. Messages are delivered in a round robin fashion to both consumers, the fast consumer processes all of its messages very quickly until its buffer is empty. At this point there are still messages awaiting to be processed in the buffer of the slow consumer thus preventing them being processed by the fast consumer. The fast consumer is therefore sitting idle when it could be processing the other messages.

To allow slow consumers, set `consumerWindowSize` on the URI to 0 (for no buffer at all). This will prevent the slow consumer from buffering any messages on the client side. Messages will remain on the server side ready to be consumed by other consumers.

Setting this to 0 can give deterministic distribution between multiple consumers on a queue.

Most of the consumers cannot be clearly identified as fast or slow consumers but are in-between. In that case, setting the value of `consumerWindowSize` to optimize performance depends on the messaging use case and requires benchmarks to find the optimal value, but a value of 1MiB is fine in most cases.

Please see the examples chapter for an example which shows how to configure ActiveMQ Artemis to prevent consumer buffering when dealing with slow consumers.

67.3. Rate limited flow control

It is also possible to control the rate at which a consumer can consume messages. This is a form of throttling and can be used to make sure that a consumer never consumes messages at a rate faster than the rate specified. This is configured using the `consumerMaxRate` URI parameter.

The rate must be a positive integer to enable this functionality and is the maximum desired message consumption rate specified in units of messages per second. Setting this to -1 disables rate limited flow control. The default value is -1.

Please see the examples chapter for a working example of limiting consumer rate.

Rate limited flow control can be used in conjunction with window based flow control. Rate limited flow control only effects how many messages a client can consume in a second and not how many messages are in its buffer. So if you had a slow rate limit and a high window based limit the clients internal buffer would soon fill up with messages.

67.4. Producer flow control

Apache ActiveMQ Artemis also can limit the amount of data sent from a client to a server to prevent the server being overwhelmed.
### 67.4.1. Window based flow control

In a similar way to consumer window based flow control, Apache ActiveMQ Artemis producers, by default, can only send messages to an address as long as they have sufficient credits to do so. The amount of credits required to send a message is given by the size of the message.

As producers run low on credits they request more from the server, when the server sends them more credits they can send more messages.

The amount of credits a producer requests in one go is known as the *window size* and it is controlled by the `producerWindowSize` URI parameter.

The window size therefore determines the amount of bytes that can be in-flight at any one time before more need to be requested - this prevents the remoting connection from getting overloaded.

#### Blocking CORE Producers

When using the CORE protocol (used by both the Artemis Core Client and Artemis JMS Client) the server will always aim give the same number of credits as have been requested. However, it is also possible to set a maximum size on any address, and the server will never send more credits to any one producer than what is available according to the address’s upper memory limit. Although a single producer will be issued more credits than available (at the time of issue) it is possible that more than 1 producer be associated with the same address and so it is theoretically possible that more credits are allocated across total producers than what is available. It is therefore possible to go over the address limit by approximately:

\[
\text{total number of producers on address} \times \text{producer window size}
\]

For example, if I have a queue called "myqueue", I could set the maximum memory size to 10MiB, and the server will control the number of credits sent to any producers which are sending any messages to myqueue such that the total messages in the queue never exceeds 10MiB.

When the address gets full, producers will block on the client side until more space frees up on the address, i.e. until messages are consumed from the queue thus freeing up space for more messages to be sent.

We call this blocking producer flow control, and it's an efficient way to prevent the server running out of memory due to producers sending more messages than can be handled at any time.

It is an alternative approach to paging, which does not block producers but instead pages messages to storage.

To configure an address with a maximum size and tell the server that you want to block producers for this address if it becomes full, you need to define an AddressSettings ([Configuring Queues Via Address Settings](#)) block for the address and specify `max-size-bytes` and `address-full-policy`.

The address block applies to all queues registered to that address, i.e. the total memory for all queues bound to that address will not exceed `max-size-bytes`. In the case of JMS topics this means the *total* memory of all subscriptions in the topic won’t exceed `max-size-bytes`. 
Here's an example:

```xml
<address-settings>
  <address-setting match="exampleQueue">
    <max-size-bytes>100000</max-size-bytes>
    <address-full-policy>BLOCK</address-full-policy>
  </address-setting>
</address-settings>
```

The above example would set the max size of the queue "exampleQueue" to be 100000 bytes and would block any producers sending to that address to prevent that max size being exceeded.

Note the policy must be set to **BLOCK** to enable blocking producer flow control.

Note that in the default configuration all addresses are set to block producers after 10 MiB of message data is in the address. This means you cannot send more than 10MiB of message data to an address without it being consumed before the producers will be blocked. If you do not want this behaviour increase the **max-size-bytes** parameter or change the address full message policy.

Producer credits are allocated from the broker to the client. Flow control credit checking (i.e. checking a producer has enough credit) is done on the client side only. It is possible for the broker to over allocate credits, like in the multiple producer scenario outlined above. It is also possible for a misbehaving client to ignore the flow control credits issued by the broker and continue sending with out sufficient credit.

**Blocking AMQP Producers**

Apache ActiveMQ Artemis ships with out of the box with 2 protocols that support flow control. Artemis CORE protocol and AMQP. Both protocols implement flow control slightly differently and therefore address full BLOCK policy behaves slightly different for clients that use each protocol respectively.

As explained earlier in this chapter the CORE protocol uses a producer window size flow control system. Where credits (representing bytes) are allocated to producers, if a producer wants to send a message it should wait until it has enough byte credits available for it to send. AMQP flow control credits are not representative of bytes but instead represent the number of messages a producer is permitted to send (regardless of the message size).

BLOCK for AMQP works mostly in the same way as the producer window size mechanism above. Artemis will issue 100 credits to a client at a time and refresh them when the clients credits reaches 30. The broker will stop issuing credits once an address is full. However, since AMQP credits represent whole messages and not bytes, it would be possible in some scenarios for an AMQP client to significantly exceed an address upper bound should the broker continue accepting messages until the clients credits are exhausted. For this reason there is an additional parameter available on address settings that specifies an upper bound on an address size in bytes. Once this upper bound is reach Artemis will start rejecting AMQP messages. This limit is the **max-size-bytes-reject-threshold**
and is by default set to -1 (or no limit). This is additional parameter allows a kind of soft and hard limit, in normal circumstances the broker will utilize the max-size-bytes parameter using flow control to put back pressure on the client, but will protect the broker by rejecting messages once the address size is reached.

### 67.4.2. Rate limited flow control

Apache ActiveMQ Artemis also allows the rate a producer can emit message to be limited, in units of messages per second. By specifying such a rate, Apache ActiveMQ Artemis will ensure that producer never produces messages at a rate higher than that specified. This is controlled by the `producerMaxRate` URL parameter.

The `producerMaxRate` must be a positive integer to enable this functionality and is the maximum desired message production rate specified in units of messages per second. Setting this to `-1` disables rate limited flow control. The default value is `-1`.

Please see the examples chapter for a working example of limiting producer rate.
Chapter 68. Plugin Support

Apache ActiveMQ Artemis is designed to allow extra functionality to be added by creating a plugin. Multiple plugins can be registered at the same time and they will be chained together and executed in the order they are registered (i.e. the first plugin registered is always executed first).

Creating a plugin is very simple. It requires:

- Implementing the ActiveMQServerPlugin interface
- Making sure the plugin is on the classpath
- Registering it with the broker either via xml or programmatically.

Only the methods that you want to add behavior for need to be implemented as all of the interface methods are default methods.

68.1. Registering a Plugin

To register a plugin with by XML you need to add the broker-plugins element at the broker.xml. It is also possible to pass configuration to a plugin using the property child element(s). These properties (zero to many) will be read and passed into the plugin’s init(Map<String, String>) operation after the plugin has been instantiated.

```xml
<broker-plugins>
   <broker-plugin class-name="some.plugin.UserPlugin">
      <property key="property1" value="val_1" />
      <property key="property2" value="val_2" />
   </broker-plugin>
</broker-plugins>
```

68.2. Registering a Plugin Programmatically

For registering a plugin programmatically you need to call the registerBrokerPlugin() method and pass in a new instance of your plugin. In the example below assuming your plugin is called UserPlugin, registering it looks like the following:

```java
Configuration config = new ConfigurationImpl();
...
config.registerBrokerPlugin(new UserPlugin());
```
68.3. Using the **LoggingActiveMQServerPlugin**

The **LoggingActiveMQServerPlugin** logs specific broker events.

You can select which events are logged by setting the following configuration properties to **true**.

<table>
<thead>
<tr>
<th>Property</th>
<th>Trigger Event</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOG_CONNECTION_EVENTS</strong></td>
<td>Connection is created/destroy.</td>
<td>false</td>
</tr>
<tr>
<td><strong>LOG_SESSION_EVENTS</strong></td>
<td>Session is created/closed.</td>
<td>false</td>
</tr>
<tr>
<td><strong>LOG_CONSUMER_EVENTS</strong></td>
<td>Consumer is created/closed.</td>
<td>false</td>
</tr>
<tr>
<td><strong>LOG_DELIVERING_EVENTS</strong></td>
<td>Message is delivered to a consumer and when a message is acknowledged by a consumer.</td>
<td>false</td>
</tr>
<tr>
<td><strong>LOG_SENDING_EVENTS</strong></td>
<td>When a message has been sent to an address and when a message has been routed within the broker.</td>
<td>false</td>
</tr>
<tr>
<td><strong>LOG_INTERNAL_EVENTS</strong></td>
<td>When a queue created/destroyed, when a message is expired, when a bridge is deployed and when a critical failure occurs.</td>
<td>false</td>
</tr>
<tr>
<td><strong>LOG_ALL_EVENTS</strong></td>
<td>Includes all the above events.</td>
<td>false</td>
</tr>
</tbody>
</table>

By default the **LoggingActiveMQServerPlugin** will not log any information. The logging is activated by setting one (or a selection) of the above configuration properties to **true**.

To configure the plugin, you can add the following configuration to the broker. In the example below both **LOG_DELIVERING_EVENTS** and **LOG_SENDING_EVENTS** will be logged by the broker.

```xml
<broker-plugins>
  <broker-plugin class-name="org.apache.activemq.artemis.core.server.plugin.impl.LoggingActiveMQServerPlugin">
    <property key="LOG_DELIVERING_EVENTS" value="true" />
    <property key="LOG_SENDING_EVENTS" value="true" />
  </broker-plugin>
</broker-plugins>
```

Most events in the **LoggingActiveMQServerPlugin** follow a **beforeX** and **afterX** notification pattern (e.g. **beforeCreateConsumer()** and **afterCreateConsumer()**).

At Log Level **INFO**, the **LoggingActiveMQServerPlugin** logs an entry when an **afterX** notification occurs. By setting the logger **org.apache.activemq.artemis.core.server.plugin.impl** to **DEBUG**, log
entries are generated for both beforeX and afterX notifications. Log level DEBUG will also log more information for a notification when available.

### 68.4. Using the NotificationActiveMQServerPlugin

The NotificationActiveMQServerPlugin can be configured to send extra notifications for specific broker events.

You can select which notifications are sent by setting the following configuration properties to true.

<table>
<thead>
<tr>
<th>Property</th>
<th>Property Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND_CONNECTION_NOTIFICATIONS</td>
<td>Sends a notification when a Connection is created/destroy.</td>
<td>false</td>
</tr>
<tr>
<td>SEND_SESSION_NOTIFICATIONS</td>
<td>Sends a notification when a Session is created/closed.</td>
<td>false</td>
</tr>
<tr>
<td>SEND_ADDRESS_NOTIFICATIONS</td>
<td>Sends a notification when an Address is added/removed.</td>
<td>false</td>
</tr>
<tr>
<td>SEND_DELIVERED_NOTIFICATIONS</td>
<td>Sends a notification when message is delivered to a consumer.</td>
<td>false</td>
</tr>
<tr>
<td>SEND_EXPIRED_NOTIFICATIONS</td>
<td>Sends a notification when message has been expired by the broker.</td>
<td>false</td>
</tr>
</tbody>
</table>

By default the NotificationActiveMQServerPlugin will not send any notifications. The plugin is activated by setting one (or a selection) of the above configuration properties to true.

To configure the plugin, you can add the following configuration to the broker. In the example below both SEND_CONNECTION_NOTIFICATIONS and SEND_SESSION_NOTIFICATIONS will be sent by the broker.

```xml
<broker-plugins>
    <broker-plugin class-name="org.apache.activemq.artemis.core.server.plugin.impl.NotificationActiveMQServerPlugin">
        <property key="SEND_CONNECTION_NOTIFICATIONS" value="true" />
        <property key="SEND_SESSION_NOTIFICATIONS" value="true" />
    </broker-plugin>
</broker-plugins>
```

### 68.5. Using the BrokerMessageAuthorizationPlugin

The BrokerMessageAuthorizationPlugin filters messages sent to consumers based on if they have a role that matches the value specified in a message property.
You can select which property will be used to specify the required role for consuming a message by setting the following configuration.

<table>
<thead>
<tr>
<th>Property</th>
<th>Property Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLE_PROPERTY</td>
<td>Property name used to determine the role required to consume a message.</td>
<td>requiredRole.</td>
</tr>
</tbody>
</table>

If the message does not have a property matching the configured ROLE_PROPERTY then the message will be sent to any consumer.

To configure the plugin, you can add the following configuration to the broker. In the example below ROLE_PROPERTY is set to permissions when that property is present messages will only be sent to consumers with a role matching its value.

```xml
<broker-plugins>
  <broker-plugin class-name="org.apache.activemq.artemis.core.server.plugin.impl.BrokerMessageAuthorizationPlugin">
    <property key="ROLE_PROPERTY" value="permissions" />
  </broker-plugin>
</broker-plugins>
```

### 68.6. Using the ConnectionPeriodicExpiryPlugin

The ConnectionPeriodicExpiryPlugin will implement a global expiry (and disconnect) for connections that live longer than periodSeconds on a matching acceptor basis.

This plugin can be useful when credential rotation or credential validation must be enforced at regular intervals as authentication will be enforced on reconnect.

The plugin requires the configuration of the acceptorMatchRegex to determine the acceptors to monitor. It is typical to separate client acceptors and federation or cluster acceptors such that only client connections will be subject to periodic expiry. The acceptorMatchRegex must be configured to match the name of the acceptor(s) whose connections will be subject to periodic expiry.

<table>
<thead>
<tr>
<th>Property</th>
<th>Property Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>acceptorMatchRegex</td>
<td>the regular expression used to match against the names of acceptors to monitor</td>
<td></td>
</tr>
<tr>
<td>periodSeconds</td>
<td>the max period, in seconds, that a connection can last</td>
<td>900 seconds (15 minutes)</td>
</tr>
<tr>
<td>Property</td>
<td>Property Description</td>
<td>Default Value</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>accuracyWindowSeconds</td>
<td>determines how often we check connections for expiry and also provides an upper bound on the random seconds we use to schedule a disconnect. Using a random second will potentially avoid many reconnects occurring at the exact same time. It must be positive value &gt; 0</td>
<td>30 seconds</td>
</tr>
</tbody>
</table>

The plugin can be configured via xml in the normal broker-plugin way:

```xml
<broker-plugins>
  <broker-plugin class-name="org.apache.activemq.artemis.core.server.plugin.impl.ConnectionPeriodicExpiryPlugin">
    <property key="acceptorMatchRegex" value="netty-client-acceptor" />
  </broker-plugin>
</broker-plugins>
```
Chapter 69. Intercepting Operations

Apache ActiveMQ Artemis supports *interceptors* to intercept packets entering and exiting the server. Incoming and outgoing interceptors are be called for any packet entering or exiting the server respectively. This allows custom code to be executed, e.g. for auditing packets, filtering or other reasons. Interceptors can change the packets they intercept. This makes interceptors powerful, but also potentially dangerous.

69.1. Implementing The Interceptors

All interceptors are protocol specific.

An interceptor for the core protocol must implement the interface `Interceptor`:

```java
package org.apache.activemq.artemis.api.core.interceptor;

public interface Interceptor {
    boolean intercept(Packet packet, RemotingConnection connection) throws ActiveMQException;
}
```

For stomp protocol an interceptor must implement the interface `StompFrameInterceptor`:

```java
package org.apache.activemq.artemis.core.protocol.stomp;

public interface StompFrameInterceptor extends BaseInterceptor<StompFrame> {
    boolean intercept(StompFrame stompFrame, RemotingConnection connection);
}
```

Likewise for MQTT protocol, an interceptor must implement the interface `MQTIInterceptor`:

```java
package org.apache.activemq.artemis.core.protocol.mqtt;

public interface MQTTInterceptor extends BaseInterceptor<MqttMessage> {
    boolean intercept(MqttMessage mqttMessage, RemotingConnection connection);
}
```

The returned boolean value is important:

- if `true` is returned, the process continues normally
- if `false` is returned, the process is aborted, no other interceptors will be called and the packet will not be processed further by the server.
### 69.2. Configuring The Interceptors

Both incoming and outgoing interceptors are configured in `broker.xml`:

```xml
<remoting-incoming-interceptors>
  <class-name>org.apache.activemq.artemis.jms.example.LoginInterceptor</class-name>
  <class-name>org.apache.activemq.artemis.jms.example.AdditionalPropertyInterceptor</class-name>
</remoting-incoming-interceptors>

<remoting-outgoing-interceptors>
  <class-name>org.apache.activemq.artemis.jms.example.LogoutInterceptor</class-name>
  <class-name>org.apache.activemq.artemis.jms.example.AdditionalPropertyInterceptor</class-name>
</remoting-outgoing-interceptors>
```

See the documentation on adding runtime dependencies to understand how to make your interceptor available to the broker.

### 69.3. Interceptors on the Client Side

The interceptors can also be run on the Apache ActiveMQ Artemis client side to intercept packets either sent by the client to the server or by the server to the client. This is done by adding the interceptor to the `ServerLocator` with the `addIncomingInterceptor(Interceptor)` or `addOutgoingInterceptor(Interceptor)` methods.

As noted above, if an interceptor returns `false` then the sending of the packet is aborted which means that no other interceptors are be called and the packet is not be processed further by the client. Typically this process happens transparently to the client (i.e. it has no idea if a packet was aborted or not). However, in the case of an outgoing packet that is sent in a **blocking** fashion a `ActiveMQException` will be thrown to the caller. The exception is thrown because blocking sends provide reliability and it is considered an error for them not to succeed. **Blocking** sends occurs when, for example, an application invokes `setBlockOnNonDurableSend(true)` or `setBlockOnDurableSend(true)` on its `ServerLocator` or if an application is using a JMS connection factory retrieved from JNDI that has either `block-on-durable-send` or `block-on-non-durable-send` set to `true`. Blocking is also used for packets dealing with transactions (e.g. commit, roll-back, etc.). The `ActiveMQException` thrown will contain the name of the interceptor that returned false.

As on the server, the client interceptor classes (and their dependencies) must be added to the classpath to be properly instantiated and invoked.

### 69.4. Examples

See the following examples which show how to use interceptors:

- **Interceptor**
- **Interceptor AMQP**
• Interceptor Client
• Interceptor MQTT
Chapter 70. Configuration Reload

The system will perform a periodic check on the configuration files, configured by `configuration-file-refresh-period`, with the default at 5000, in milliseconds. These checks can be disabled by specifying -1.

Note that the Log4J2 configuration has its own reload mechanism, configured via its own log4j2.properties file. See Logging configuration reload for more detail.

Once the configuration file is changed (broker.xml) the following modules will be reloaded automatically:

- Address Settings
- Security Settings
- Diverts
- Addresses & Queues
- Bridges

If using modularised broker.xml ensure you also read reloading modular configuration files.

Addresses, queues and diverts can be removed automatically when removed from the configuration:

**config-delete-addresses**
- **OFF** (default) - will not remove the address upon config reload. Messages left in the attached queues will be left intact.
- **FORCE** - will remove the address and its queues upon config reload. Messages left in the attached queues will be **lost**.

**config-delete-queues**
- **OFF** (default) - will not remove the queues upon config reload. Messages left in the queues will be left intact.
- **FORCE** - will remove the queues. Messages left in the queues will be **lost**.

**config-delete-diverts**
- **OFF** (default) - will not remove the diverts upon config reload.
- **FORCE** - will remove the diverts upon config reload.

By default, all settings are **OFF**, so that addresses, queues and diverts aren’t removed upon configuration reload, reducing the risk of losing messages.

Addresses, queues and diverts no longer present in the configuration file can be removed manually via the web interface, CLI, or JMX management operations.
70.1. Reloadable Parameters

The broker configuration file has 2 main parts, `<core>` and `<jms>`. Some of the parameters in the 2 parts are monitored and, if modified, reloaded into the broker at runtime.

Elements under `<jms>` are deprecated. Users are encouraged to use `<core>` configuration entities.

Most parameters reloaded take effect immediately after reloading. However there are some that won’t take any effect unless you restarting the broker. Such parameters are specifically indicated in the following text.

70.1.1. `<core>`

`<security-settings>`

- `<security-setting>` element

Changes to any `<security-setting>` elements will be reloaded. Each `<security-setting>` defines security roles for a matched address.

- The `match` attribute

  This attribute defines the address for which the security-setting is defined. It can take wildcards such as ‘#’ and ‘*’.

- The `<permission>` sub-elements

  Each `<security-setting>` can have a list of `<permission>` elements, each of which defines a specific permission-roles mapping. Each permission has 2 attributes ‘type’ and ‘roles’. The ‘type’ attribute defines the type of operation allowed, the ‘roles’ defines which roles are allowed to perform such operation. Refer to the user's manual for a list of operations that can be defined.

Once loaded the security-settings will take effect immediately. Any new clients will subject to the new security settings. Any existing clients will subject to the new settings as well, as soon as they performs a new security-sensitive operation.

Below lists the effects of adding, deleting and updating of an element/attribute within the `<security-settings>` element, whether a change can be done or can't be done.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Add</th>
<th>Delete</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;security-settings&gt;</code></td>
<td>X* (at most one element is allowed)</td>
<td>Deleting it means delete the whole security settings from the running broker.</td>
<td>N/A*</td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Delete</td>
<td>Update</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td><code>&lt;security-setting&gt;</code></td>
<td>Adding one element means adding a new set of security roles for an address in the running broker</td>
<td>Deleting one element means removing a set of security roles for an address in the running broker</td>
<td>Updating one element means updating the security roles for an address (if match attribute is not changed), or means removing the old match address settings and adding a new one (if match attribute is changed)</td>
</tr>
<tr>
<td>attribute <code>match</code></td>
<td>N/A*</td>
<td>X*</td>
<td>Changing this value is same as deleting the whole <code>&lt;security-setting&gt;</code> with the old match value and adding <code>&lt;/security-setting&gt;</code></td>
</tr>
<tr>
<td><code>&lt;permission&gt;</code></td>
<td>Adding one means adding a new permission definition to runtime broker</td>
<td>Deleting a permission from the runtime broker</td>
<td>Updating a permission-roles in the runtime broker</td>
</tr>
<tr>
<td>attribute <code>type</code></td>
<td>N/A*</td>
<td>X*</td>
<td>Changing the type value means remove the permission of the old one and add the permission of this type to the running broker.</td>
</tr>
<tr>
<td>attribute <code>roles</code></td>
<td>N/A*</td>
<td>X*</td>
<td>Changing the ‘roles’ value means updating the permission’s allowed roles to the running broker</td>
</tr>
</tbody>
</table>

- N/A means this operation is not applicable.
- X means this operation is not allowed.

### `<address-settings>`

- `<address-settings>` element

Changes to elements under `<address-settings>` will be reloaded into runtime broker. It contains a list of `<address-setting>` elements.

- `<address-setting>` element
Each address-setting element has a ‘match’ attribute that defines an address pattern for which this address-setting is defined. It also has a list of sub-elements used to define the properties of a matching address.

Parameters reloaded in this category will take effect immediately after reloading. The effect of deletion of Address’s and Queue’s, not auto created is controlled by parameter `config-delete-addresses` and `config-delete-queues` as described in the doc.

Below lists the effects of adding, deleting and updating of an element/attribute within the address-settings element, whether a change can be done or can’t be done.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Add</th>
<th>Delete</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;address-settings&gt;</code></td>
<td>X(at most one element is allowed)</td>
<td>Deleting it means delete the whole address settings from the running broker</td>
<td>N/A</td>
</tr>
<tr>
<td><code>&lt;address-setting&gt;</code></td>
<td>Adding one element means adding a set of address-setting for a new address in the running broker</td>
<td>Deleting one means removing a set of address-setting for an address in the running broker</td>
<td>Updating one element means updating the address setting for an address (if match attribute is not changed), or means removing the old match address settings and adding a new one (if match attribute is changed)</td>
</tr>
<tr>
<td>attribute <code>match</code></td>
<td>N/A</td>
<td>X</td>
<td>Changing this value is same as deleting the whole <code>&lt;address-setting&gt;</code> with the old match value and adding a new one with the new match value. <code>/address-setting&gt;</code></td>
</tr>
<tr>
<td><code>&lt;dead-letter-address&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>Removing the configured dead-letter-address address from running broker.</td>
<td>The dead letter address of the matching address will be updated after reloading</td>
</tr>
<tr>
<td><code>&lt;expiry-address&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>Removing the configured expiry address from running broker.</td>
<td>The expiry address of the matching address will be updated after reloading</td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Delete</td>
<td>Update</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;expiry-delay&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured expiry-delay will be removed from running broker.</td>
<td>The expiry-delay for the matching address will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;redelivery-delay&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured redelivery-delay will be removed from running broker.</td>
<td>The redelivery-delay for the matching address will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;redelivery-delay-multiplier&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured redelivery-delay-multiplier will be removed from running broker.</td>
<td>The redelivery-delay-multiplier will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;max-redelivery-delay&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured max-redelivery-delay will be removed from running broker.</td>
<td>The max-redelivery-delay will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;max-delivery-attempts&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured max-delivery-attempts will be removed from running broker.</td>
<td>The max-delivery-attempts will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;max-size-bytes&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured max-size-bytes will be removed from running broker.</td>
<td>The max-size-bytes will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;page-size-bytes&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured page-size-bytes will be removed from running broker.</td>
<td>The page-size-bytes will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;address-full-policy&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured address-full-policy will be removed from running broker.</td>
<td>The address-full-policy will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;message-counter-history-day-limit&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured message-counter-history-day-limit will be removed from running broker.</td>
<td>The message-counter-history-day-limit will be updated after reloading.</td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Delete</td>
<td>Update</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;last-value-queue&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured last-value-queue will be removed from running broker after reloading (no longer a last value queue).</td>
<td>The last-value-queue will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;redistribution-delay&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured redistribution-delay will be removed from running broker after reloading.</td>
<td>The redistribution-delay will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;send-to-dla-on-no-route&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured send-to-dla-on-no-route will be removed from running broker after reloading.</td>
<td>The send-to-dla-on-no-route will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;slow-consumer-threshold&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured slow-consumer-threshold will be removed from running broker after reloading.</td>
<td>The slow-consumer-threshold will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;slow-consumer-policy&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured slow-consumer-policy will be removed from running broker after reloading.</td>
<td>The slow-consumer-policy will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;slow-consumer-check-period&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured slow-consumer-check-period will be removed from running broker after reloading. (meaning the slow consumer checker thread will be cancelled)</td>
<td>The slow-consumer-check-period will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;auto-create-queues&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured auto-create-queues will be removed from running broker after reloading.</td>
<td>The auto-create-queues will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;auto-delete-queues&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured auto-delete-queues will be removed from running broker after reloading.</td>
<td>The auto-delete-queues will be updated after reloading.</td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Delete</td>
<td>Update</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;config-delete-queues&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured config-delete-queues will be removed from running broker after reloading.</td>
<td>The config-delete-queues will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;auto-create-addresses&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured auto-create-addresses will be removed from running broker after reloading.</td>
<td>The auto-create-addresses will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;auto-delete-addresses&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured auto-delete-addresses will be removed from running broker after reloading.</td>
<td>The auto-delete-addresses will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;config-delete-addresses&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured config-delete-addresses will be removed from running broker after reloading.</td>
<td>The config-delete-addresses will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;management-browse-page-size&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured management-browse-page-size will be removed from running broker after reloading.</td>
<td>The management-browse-page-size will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;default-purge-on-no-consumers&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured default-purge-on-no-consumers will be removed from running broker after reloading.</td>
<td>The default-purge-on-no-consumers will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;default-max-consumers&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured default-max-consumers will be removed from running broker after reloading.</td>
<td>The default-max-consumers will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;default-queue-routing-type&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured default-queue-routing-type will be removed from running broker after reloading.</td>
<td>The default-queue-routing-type will be updated after reloading.</td>
</tr>
<tr>
<td><code>&lt;default-address-routing-type&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>The configured default-address-routing-type will be removed from running broker after reloading.</td>
<td>The default-address-routing-type will be updated after reloading.</td>
</tr>
</tbody>
</table>
All `<divert>` elements will be reloaded. Each `<divert>` element has a ‘name’ and several sub-elements that defines the properties of a divert.

Existing diverts get undeployed if you delete their `<divert>` element.

Below lists the effects of adding, deleting and updating of an element/attribute within the `diverts` element, whether a change can be done or can’t be done.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Add</th>
<th>Delete</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;diverts&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>Deleting it means delete (undeploy) all diverts in running broker.</td>
<td>N/A</td>
</tr>
<tr>
<td><code>&lt;divert&gt;</code></td>
<td>Adding a new divert. It will be deployed after reloading</td>
<td>Deleting it means the divert will be undeployed after reloading</td>
<td>No effect on the deployed divert (unless restarting broker, in which case the divert will be redeployed)</td>
</tr>
<tr>
<td>attribute name</td>
<td>N/A</td>
<td>X</td>
<td>No effect on the deployed divert (unless restarting broker, in which case the divert has the transformer class)</td>
</tr>
<tr>
<td><code>&lt;transformer-class-name&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>No effect on the deployed divert (unless restarting broker, in which case the divert will be deployed without the transformer class)</td>
<td>No effect on the deployed divert (unless restarting broker, in which case the divert has the transformer class)</td>
</tr>
<tr>
<td><code>&lt;exclusive&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>No effect on the deployed divert (unless restarting broker)</td>
<td>No effect on the deployed divert (unless restarting broker)</td>
</tr>
<tr>
<td><code>&lt;routing-name&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>No effect on the deployed divert (unless restarting broker)</td>
<td>No effect on the deployed divert (unless restarting broker)</td>
</tr>
<tr>
<td><code>&lt;address&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>No effect on the deployed divert (unless restarting broker)</td>
<td>No effect on the deployed divert (unless restarting broker)</td>
</tr>
<tr>
<td><code>&lt;forwarding-address&gt;</code></td>
<td>X (no more than one can be present)</td>
<td>No effect on the deployed divert (unless restarting broker)</td>
<td>No effect on the deployed divert (unless restarting broker)</td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Delete</td>
<td>Update</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------</td>
<td>--------------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>&lt;filter&gt;</td>
<td>X (no more than one can be present)</td>
<td>No effect on the deployed divert. (unless restarting broker)</td>
<td>No effect on the deployed divert. (unless restarting broker)</td>
</tr>
<tr>
<td>&lt;routing-type&gt;</td>
<td>X (no more than one can be present)</td>
<td>No effect on the deployed divert. (unless restarting broker)</td>
<td>No effect on the deployed divert. (unless restarting broker)</td>
</tr>
</tbody>
</table>

**<addresses>**

The `<addresses>` element contains a list `<address>` elements. Once changed, all `<address>` elements in `<addresses>` will be reloaded.

- Once reloaded, all new addresses (as well as the pre-configured queues) will be deployed to the running broker and all those that are missing from the configuration will be undeployed.
- Parameters reloaded in this category will take effect immediately after reloading. The effect of deletion of Address’s and Queue’s, not auto created is controlled by parameter `config-delete-addresses` and `config-delete-queues` as described in this doc.

Below lists the effects of adding, deleting and updating of an element/attribute within the `<addresses>` element, whether a change can be done or can’t be done.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Add</th>
<th>Delete</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;addresses&gt;</code></td>
<td>X (no more than one is present)</td>
<td>Deleting it means delete (undeploy) all diverts in running broker</td>
<td>N/A</td>
</tr>
<tr>
<td><code>&lt;address&gt;</code></td>
<td>A new address will be deployed in the running broker</td>
<td>The corresponding address will be undeployed.</td>
<td>N/A</td>
</tr>
<tr>
<td>attribute name</td>
<td>N/A</td>
<td>X</td>
<td>After reloading the address of the old name will be undeployed and the new will be deployed.</td>
</tr>
<tr>
<td><code>&lt;anycast&gt;</code></td>
<td>X (no more than one is present)</td>
<td>The anycast routing type will be undeployed from this address, as well as its containing queues after reloading</td>
<td>N/A</td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Delete</td>
<td>Update</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------</td>
<td>---------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>&lt;queue&gt;(under anycast&gt;)</td>
<td>An anycast queue will be deployed after reloading</td>
<td>The anycast queue will be undeployed</td>
<td>For updating queues please see next section &lt;queue&gt;</td>
</tr>
<tr>
<td>&lt;multicast&gt;</td>
<td>X(no more than one is present)</td>
<td>The multicast routing type will be undeployed from this address, as well as its containing queues after reloading</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;queue&gt;(under multicast&gt;)</td>
<td>A multicast queue will be deployed after reloading</td>
<td>The multicast queue will be undeployed</td>
<td>For updating queues please see next section &lt;queue&gt;</td>
</tr>
</tbody>
</table>

Changes to any <queue> elements will be reloaded to the running broker.

- Once reloaded, all new queues will be deployed to the running broker and all queues that are missing from the configuration will be undeployed.
- Parameters reloaded in this category will take effect immediately after reloading. The effect of deletion of Address's and Queue's, not auto created is controlled by parameter `config-delete-addresses` and `config-delete-queues` as described in this doc.

Below lists the effects of adding, deleting and updating of an element/attribute within the <queue> element, and whether a change can be done or can’t be done.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Add</th>
<th>Delete</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;queue&gt;</td>
<td>A new queue is deployed after reloading</td>
<td>The queue will be undeployed after reloading.</td>
<td>N/A</td>
</tr>
<tr>
<td>attribute name</td>
<td>N/A</td>
<td>X</td>
<td>A queue with new name will be deployed and the queue with old name will be undeployed after reloading (see Note above).</td>
</tr>
<tr>
<td>attribute max-consumers</td>
<td>If max-consumers &gt; current consumers max-consumers will update on reload</td>
<td>max-consumers will be set back to the default -1</td>
<td>If max-consumers &gt; current consumers max-consumers will update on reload</td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Delete</td>
<td>Update</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>attribute purge-on-no-consumers</td>
<td>On reload purge-on-no-consumers will be updated</td>
<td>Will be set back to the default false</td>
<td>On reload purge-on-no-consumers will be updated</td>
</tr>
<tr>
<td>attribute enabled</td>
<td>On reload enabled will be updated</td>
<td>Will be set back to the default true</td>
<td>On reload enabled will be updated</td>
</tr>
<tr>
<td>attribute exclusive</td>
<td>On reload exclusive will be updated</td>
<td>Will be set back to the default false</td>
<td>On reload exclusive will be updated</td>
</tr>
<tr>
<td>attribute group-rebalance</td>
<td>On reload group-rebalance will be updated</td>
<td>Will be set back to the default false</td>
<td>On reload group-rebalance will be updated</td>
</tr>
<tr>
<td>attribute group-rebalance-pause-dispatch</td>
<td>On reload group-rebalance-pause-dispatch will be updated</td>
<td>Will be set back to the default false</td>
<td>On reload group-rebalance-pause-dispatch will be updated</td>
</tr>
<tr>
<td>attribute group-buckets</td>
<td>On reload group-buckets will be updated</td>
<td>Will be set back to the default -1</td>
<td>On reload group-buckets will be updated</td>
</tr>
<tr>
<td>attribute group-first-key</td>
<td>On reload group-first-key will be updated</td>
<td>Will be set back to the default null</td>
<td>On reload group-first-key will be updated</td>
</tr>
<tr>
<td>attribute last-value</td>
<td>On reload last-value will be updated</td>
<td>Will be set back to the default false</td>
<td>On reload last-value will be updated</td>
</tr>
<tr>
<td>attribute last-value-key</td>
<td>On reload last-value-key will be updated</td>
<td>Will be set back to the default null</td>
<td>On reload last-value-key will be updated</td>
</tr>
<tr>
<td>attribute non-destructive</td>
<td>On reload non-destructive will be updated</td>
<td>Will be set back to the default false</td>
<td>On reload non-destructive will be updated</td>
</tr>
<tr>
<td>attribute consumers-before-dispatch</td>
<td>On reload consumers-before-dispatch will be updated</td>
<td>Will be set back to the default 0</td>
<td>On reload consumers-before-dispatch will be updated</td>
</tr>
<tr>
<td>attribute delay-before-dispatch</td>
<td>On reload delay-before-dispatch will be updated</td>
<td>Will be set back to the default -1</td>
<td>On reload delay-before-dispatch will be updated</td>
</tr>
<tr>
<td>attribute ring-size</td>
<td>On reload ring-size will be updated</td>
<td>Will be set back to the default -1</td>
<td>On reload ring-size will be updated</td>
</tr>
<tr>
<td>&lt;filter&gt;</td>
<td>The filter will be added after reloading</td>
<td>The filter will be removed after reloading</td>
<td>The filter will be updated after reloading</td>
</tr>
<tr>
<td>&lt;durable&gt;</td>
<td>The queue durability will be set to the given value after reloading</td>
<td>The queue durability will be set to the default true after reloading</td>
<td>The queue durability will be set to the new value after reloading</td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Delete</td>
<td>Update</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;user&gt;</code></td>
<td>The queue user will be set to the given value after reloading</td>
<td>The queue user will be set to the default <code>null</code> after reloading</td>
<td>The queue user will be set to the new value after reloading</td>
</tr>
</tbody>
</table>

70.1.2. `<jms>` *( Deprecated )*  

70.1.3. `<queues>` *( Deprecated )*
Chapter 71. Detecting Slow Consumers

In this section we will discuss how Apache ActiveMQ Artemis can be configured to deal with slow consumers. A slow consumer with a server-side queue (e.g. JMS topic subscriber) can pose a significant problem for broker performance. If messages build up in the consumer's server-side queue then memory will begin filling up and the broker may enter paging mode which would impact performance negatively. However, criteria can be set so that consumers which don't acknowledge messages quickly enough can potentially be disconnected from the broker, which in the case of a non-durable JMS subscriber, would allow the broker to remove the subscription and all of its messages freeing up valuable server resources.

71.1. Required Configuration

By default the server will not detect slow consumers. If slow consumer detection is desired then see address model chapter for more details on the required address settings.

The calculation to determine whether or not a consumer is slow only inspects the number of messages a particular consumer has acknowledged. It doesn't take into account whether or not flow control has been enabled on the consumer, whether or not the consumer is streaming a large message, etc. Keep this in mind when configuring slow consumer detection.

Please note that slow consumer checks are performed using the scheduled thread pool and that each queue on the broker with slow consumer detection enabled will cause a new entry in the internal java.util.concurrent.ScheduledThreadPoolExecutor instance. If there are a high number of queues and the slow-consumer-check-period is relatively low then there may be delays in executing some of the checks. However, this will not impact the accuracy of the calculations used by the detection algorithm. See thread pooling for more details about this pool.

71.2. Example

See the slow consumer example which shows how to detect a slow consumer with Apache ActiveMQ Artemis.
Chapter 72. Critical Analysis of the broker

There are a few things that can go wrong on a production environment:

- Bugs, for more than we try they still happen! We always try to correct them, but that’s the only constant in software development.
- IO Errors, disks and hardware can go bad
- Memory issues, the CPU can go crazy by another process

For cases like this, we added a protection to the broker to shut itself down when bad things happen.

This is a feature I hope you won't need it, think it as a safeguard:

We measure time response in places like:

- Queue delivery (add to the queue)
- Journal storage
- Paging operations

If the response time goes beyond a configured timeout, the broker is considered unstable and an action will be taken to either shutdown the broker or halt the VM.

You can use these following configuration options on broker.xml to configure how the critical analysis is performed.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical-analyzer</td>
<td>Enable or disable the critical analysis (default true)</td>
</tr>
<tr>
<td>critical-analyzer-timeout</td>
<td>Timeout used to do the critical analysis (default 120000 milliseconds)</td>
</tr>
<tr>
<td>critical-analyzer-check-period</td>
<td>Time used to check the response times (default half of critical-analyzer-timeout)</td>
</tr>
<tr>
<td>critical-analyzer-policy</td>
<td>Should the server log, be halted or shutdown upon failures (default LOG)</td>
</tr>
</tbody>
</table>

The default for critical-analyzer-policy is LOG, however the generated broker.xml will have it set to HALT. That is because we cannot halt the VM if you are embedding ActiveMQ Artemis into an application server or on a multi tenant environment.

The broker on the distribution will then have it set to HALT, but if you use it in any other way the default will be LOG.

72.1. What to Expect

- You will see some logs
If you have critical-analyzer-policy=HALT

```
[Artemis Critical Analyzer] 18:10:00,831 ERROR
[org.apache.activemq.artemis.core.server] AMQ224079: The process for the virtual
machine will be killed, as component
org.apache.activemq.artemis.tests.integration.critical.CriticalSimpleTest$2@5af97850
is not responsive
```

While if you have critical-analyzer-policy=SHUTDOWN

```
[org.apache.activemq.artemis.core.server] AMQ224080: The server process will now be
stopped, as component
org.apache.activemq.artemis.tests.integration.critical.CriticalSimpleTest$2@5af97850
is not responsive
```

Or if you have critical-analyzer-policy=LOG

```
[Artemis Critical Analyzer] 18:11:52,145 WARN
[org.apache.activemq.artemis.core.server] AMQ224081: The component
org.apache.activemq.artemis.tests.integration.critical.CriticalSimpleTest$2@5af97850
is not responsive
```

You will see a simple thread dump of the server

```
[Artemis Critical Analyzer] 18:10:00,836 WARN
[org.apache.activemq.artemis.core.server] AMQ222199: Thread dump: AMQ119001:
Generating thread dump
*******************************************************************************
===============================================================================
AMQ119002: Thread Thread[Thread-1 (ActiveMQ-scheduled-threads),5,main] name = Thread-1
(ActiveMQ-scheduled-threads) id = 19 group =
java.lang.ThreadGroup[name=main,maxpri=10]
sun.misc.Unsafe.park(Native Method)
java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2039)
java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:1088)
java.util.concurrent.ScheduledThreadPoolExecutor$DelayedWorkQueue.take(ScheduledThreadPoolExecutor.java:809)
java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1067)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1127)
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
java.lang.Thread.run(Thread.java:745)
===============================================================================
```
• The Server will be halted if configured to **HALT**

• The system will be stopped if **SHUTDOWN** is used. **Notice:** If the system is not behaving well, there is no guarantees the stop will work.
Chapter 73. Resource Manager Configuration

Apache ActiveMQ Artemis has its own Resource Manager for handling the lifespan of JTA transactions. When a transaction is started the resource manager is notified and keeps a record of the transaction and its current state. It is possible in some cases for a transaction to be started but then forgotten about. Maybe the client died and never came back. If this happens then the transaction will just sit there indefinitely.

To cope with this Apache ActiveMQ Artemis can, if configured, scan for old transactions and rollback any it finds. The default for this is 3000000 milliseconds (5 minutes), i.e. any transactions older than 5 minutes are removed. This timeout can be changed by editing the `transaction-timeout` property in `broker.xml` (value must be in milliseconds). The property `transaction-timeout-scan-period` configures how often, in milliseconds, to scan for old transactions.

Please note that Apache ActiveMQ Artemis will not unilaterally rollback any XA transactions in a prepared state - this must be heuristically rolled back via the management API if you are sure they will never be resolved by the transaction manager.
Chapter 74. Guarantees of Sends and Commits

74.1. Transaction Completion

When committing or rolling back a transaction with Apache ActiveMQ Artemis, the request to commit or rollback is sent to the server, and the call will block on the client side until a response has been received from the server that the commit or rollback was executed.

When the commit or rollback is received on the server, it will be committed to the journal, and depending on the value of the parameter `journal-sync-transactional` the server will ensure that the commit or rollback is durably persisted to storage before sending the response back to the client. If this parameter has the value `false` then commit or rollback may not actually get persisted to storage until some time after the response has been sent to the client. In event of server failure this may mean the commit or rollback never gets persisted to storage. The default value of this parameter is `true` so the client can be sure all transaction commits or rollbacks have been persisted to storage by the time the call to commit or rollback returns.

Setting this parameter to `false` can improve performance at the expense of some loss of transaction durability.

This parameter is set in `broker.xml`

74.2. Non Transactional Message Sends

If you are sending messages to a server using a non transacted session, Apache ActiveMQ Artemis can be configured to block the call to send until the message has definitely reached the server, and a response has been sent back to the client. This can be configured individually for durable and non-durable messages, and is determined by the following two URL parameters:

`blockOnDurableSend`

If this is set to `true` then all calls to send for durable messages on non transacted sessions will block until the message has reached the server, and a response has been sent back. The default value is `true`.

`blockOnNonDurableSend`

If this is set to `true` then all calls to send for non-durable messages on non transacted sessions will block until the message has reached the server, and a response has been sent back. The default value is `false`.

Setting block on sends to `true` can reduce performance since each send requires a network round trip before the next send can be performed. This means the performance of sending messages will be limited by the network round trip time (RTT) of your network, rather than the bandwidth of your network. For better performance we recommend either batching many messages sends together in a transaction since with a transactional session, only the commit / rollback blocks not every send, or, using Apache ActiveMQ Artemis’s advanced asynchronous send acknowledgements.
When the server receives a message sent from a non transactional session, and that message is durable and the message is routed to at least one durable queue, then the server will persist the message in permanent storage. If the journal parameter `journal-sync-non-transactional` is set to `true` the server will not send a response back to the client until the message has been persisted and the server has a guarantee that the data has been persisted to disk. The default value for this parameter is `true`.

### 74.3. Non Transactional Acknowledgements

If you are acknowledging the delivery of a message at the client side using a non transacted session, Apache ActiveMQ Artemis can be configured to block the call to acknowledge until the acknowledge has definitely reached the server, and a response has been sent back to the client. This is configured with the parameter `BlockOnAcknowledge`. If this is set to `true` then all calls to acknowledge on non transacted sessions will block until the acknowledge has reached the server, and a response has been sent back. You might want to set this to `true` if you want to implement a strict *at most once* delivery policy. The default value is `false`.

### 74.4. Asynchronous Send Acknowledgements

If you are using a non transacted session but want a guarantee that every message sent to the server has reached it, then, as discussed in Guarantees of Non Transactional Message Sends, you can configure Apache ActiveMQ Artemis to block the call to send until the server has received the message, persisted it and sent back a response. This works well but has a severe performance penalty - each call to send needs to block for at least the time of a network round trip (RTT) - the performance of sending is thus limited by the latency of the network, *not* limited by the network bandwidth.

Let's do a little bit of maths to see how severe that is. We'll consider a standard 1Gib ethernet network with a network round trip between the server and the client of 0.25 ms.

With a RTT of 0.25 ms, the client can send at most $\frac{1000}{0.25} = 4000$ messages per second if it blocks on each message send.

If each message is < 1500 bytes and a standard 1500 bytes MTU (Maximum Transmission Unit) size is used on the network, then a 1Gib network has a *theoretical* upper limit of $(1024 * 1024 * 1024 / 8) / 1500 = 89478$ messages per second if messages are sent without blocking! These figures aren't an exact science but you can clearly see that being limited by network RTT can have serious effect on performance.

To remedy this, Apache ActiveMQ Artemis provides an advanced new feature called *asynchronous send acknowledgements*. With this feature, Apache ActiveMQ Artemis can be configured to send messages without blocking in one direction and asynchronously getting acknowledgement from the server that the messages were received in a separate stream. By de-coupling the send from the acknowledgement of the send, the system is not limited by the network RTT, but is limited by the network bandwidth. Consequently better throughput can be achieved than is possible using a blocking approach, while at the same time having absolute guarantees that messages have
successfully reached the server.

The window size for send acknowledgements is determined by the confirmation-window-size parameter on the connection factory or client session factory. Please see Client Failover for more info on this.

To use the feature using the core API, you implement the interface org.apache.activemq.artemis.api.core.client.SendAcknowledgementHandler and set a handler instance on your ClientSession.

Then, you just send messages as normal using your ClientSession, and as messages reach the server, the server will send back an acknowledgement of the send asynchronously, and some time later you are informed at the client side by Apache ActiveMQ Artemis calling your handler's sendAcknowledged(ClientMessage message) method, passing in a reference to the message that was sent.

To enable asynchronous send acknowledgements you must make sure confirmationWindowSize is set to a positive integer value, e.g. 10MiB

Please see the examples chapter for a full working example.
Chapter 75. Graceful Server Shutdown

In certain circumstances an administrator might not want to disconnect all clients immediately when stopping the broker. In this situation the broker can be configured to shutdown gracefully using the `graceful-shutdown-enabled` boolean configuration parameter.

When the `graceful-shutdown-enabled` configuration parameter is `true` and the broker is shutdown it will first prevent any additional clients from connecting and then it will wait for any existing connections to be terminated by the client before completing the shutdown process. The default value is `false`.

Of course, it’s possible a client could keep a connection to the broker indefinitely effectively preventing the broker from shutting down gracefully. To deal with this situation the `graceful-shutdown-timeout` configuration parameter is available. This tells the broker (in milliseconds) how long to wait for all clients to disconnect before forcefully disconnecting the clients and proceeding with the shutdown process. The default value is `-1` which means the broker will wait indefinitely for clients to disconnect.
Chapter 76. Embedded Web Server

Apache ActiveMQ Artemis embeds the Jetty web server. Its main purpose is to host the Management Console. However, it can also host other web applications.

76.1. Configuration

The embedded Jetty instance is configured in `etc/bootstrap.xml` via the `web` element, e.g.:

```xml
<web path="web">
  <binding uri="http://localhost:8161">
    <app url="activemq-branding" war="activemq-branding.war"/>
    <app url="artemis-plugin" war="artemis-plugin.war"/>
    <app url="console" war="console.war"/>
  </binding>
</web>
```

76.1.1. Web

The `web` element has the following attributes:

- **path**
  The name of the subdirectory in which to find the web application archives (i.e. WAR files). This is a subdirectory of the broker's home or instance directory.

- **customizer**
  The name of customizer class to load.

- **rootRedirectLocation**
  The location to redirect the requests with the root target.

- **webContentEnabled**
  Whether or not the content included in the web folder of the home and the instance directories is accessible. Default is `false`.

- **maxThreads**
  The maximum number of threads the embedded web server can create to service HTTP requests. Default is `200`.

- **minThreads**
  The minimum number of threads the embedded web server will hold to service HTTP requests. Default is `8` or the value of `maxThreads` if it is lower.

- **idleThreadTimeout**
  The time to wait before terminating an idle thread from the embedded web server. Measured in milliseconds. Default is `60000`.
scanPeriod
How often to scan for changes of the key and trust store files related to a binding when the
sslAutoReload attribute value of the binding element is true, for further details see Binding.
Measured in seconds. Default is 5.

76.1.2. Binding
The web element should contain at least one binding element to configure how clients can connect to
the web-server. A binding element has the following attributes:

uri
The protocol to use (i.e. http or https) as well as the host and port on which to listen. This
attribute is required.

clientAuth
Whether or not clients should present an SSL certificate when they connect. Only applicable
when using https.

passwordCodec
The custom codec to use for unmasking the keystorePassword and trustStorePassword.

keyStorePath
The location on disk of the keystore. Only applicable when using https.

keyStorePassword
The password to the keystore. Only applicable when using https. Can be masked using ENC() syntax or by defining passwordCodec. See more in the password masking chapter.

trustStorePath
The location on disk for the truststore. Only applicable when using https.

trustStorePassword
The password to the truststore. Only applicable when using https. Can be masked using ENC() syntax or by defining passwordCodec. See more in the password masking chapter.

includedTLSProtocols
A comma separated list of included TLS protocols, ie "TLSv1,TLSv1.1,TLSv1.2". Only applicable
when using https.

excludedTLSProtocols
A comma separated list of excluded TLS protocols, ie "TLSv1,TLSv1.1,TLSv1.2". Only applicable
when using https.

includedCipherSuites
A comma separated list of included cipher suites. Only applicable when using https.

excludedCipherSuites
A comma separated list of excluded cipher suites. Only applicable when using https.
sniHostCheck
Whether or not the SNI Host name in the client request must match the common name or the subject alternative names in the server certificate. Default is true. Only applicable when using https.

sniRequired
Whether or not the client request must include an SNI Host name. Default is false. Only applicable when using https.

sslAutoReload
Whether or not the key and trust store files must be watched for changes and automatically reloaded. The watch period is controlled by the scanPeriod attribute of the web element, for further details see Web. Default is false.

76.1.3. App
Each web application should be defined in an app element inside an binding element. The app element has the following attributes:

url
The context to use for the web application.

war
The name of the web application archive on disk.

76.2. Request Log
It's also possible to configure HTTP/S request logging via the request-log element which has the following attributes:

filename
The full path of the request log. This attribute is required.

append
Whether or not to append to the existing log or truncate it. Boolean flag.

extended
Whether or not to use the extended request log format. Boolean flag. If true will use the format %{client}a - %u %t "%r" %s %O "%{Referer}i" "%{User-Agent}i". If false will use the format %{client}a - %u %t "%r" %s %O. Default is false. See the format specification for more details.

filenameDateFormat
The log file name date format.

retainDays
The number of days before rotated log files are deleted.
ignorePaths
Request paths that will not be logged. Comma delimited list.

format
Custom format to use. If set this will override extended. See the format specification for more details.

The following options were previously supported, but they were replaced by the format: logCookie, logTimeZone, logDateFormat, logLocale, logLatency, logServer, preferProxiedForAddress. All these options are now deprecated and ignored.

These attributes are essentially passed straight through to the underlying org.eclipse.jetty.server.CustomRequestLog and org.eclipse.jetty.server.RequestLogWriter instances. Default values are based on these implementations.

Here is an example configuration:

```xml
<web path="web">
  <binding uri="http://localhost:8161">
    <app url="activemq-branding" war="activemq-branding.war"/>
    <app url="artemis-plugin" war="artemis-plugin.war"/>
    <app url="console" war="console.war"/>
  </binding>
  <request-log filename="${artemis.instance}/log/http-access-yyyy_MM_dd.log" append="true" extended="true"/>
</web>
```

76.2.1. System properties

It is possible to use system properties to add or update web configuration items. If you define a system property starting with "webconfig." it will be parsed at the startup to update the web configuration.

To enable the client authentication for an existing binding with the name artemis, set the system property webconfig.bindings.artemis.clientAuth to true, i.e.

```bash
java -Dwebconfig.bindings.artemis.clientAuth=true
```

To add a new binding or app set the new binding or app attributes using their new names, i.e.

```bash
java -Dwebconfig.bindings.my-binding.uri=http://localhost:8162
java -Dwebconfig.bindings.my-binding.apps.my-app.uri=my-app
java -Dwebconfig.bindings.my-binding.apps.my-app.war=my-app.war
```

To update a binding without a name use its uri and to update an app without a name use its url, i.e.
76.3. Proxy Forwarding

The proxies and load balancers usually support X-Forwarded headers to send information altered or lost when a proxy is involved in the path of the request. Jetty supports the ForwardedRequestCustomizer customizer to handle X-Forwarded headers. Set the customizer attribute via the web element to enable the ForwardedRequestCustomizer customizer, ie:

```xml
<web path="web" customizer="org.eclipse.jetty.server.ForwardedRequestCustomizer">
  <binding uri="http://localhost:8161">
    <app url="activemq-branding" war="activemq-branding.war"/>
    <app url="artemis-plugin" war="artemis-plugin.war"/>
    <app url="console" war="console.war"/>
  </binding>
</web>
```

76.4. Management

The embedded web server can be stopped, started, or restarted via any available management interface via the stopEmbeddedWebServer, startEmbeddedWebServer, and restartEmbeddedWebServer operations on the ActiveMQServerControl respectively.
Chapter 77. Logging

Apache ActiveMQ Artemis uses the SLF4J logging facade for logging, with the broker assembly providing Log4J 2 as the logging implementation. This is configurable via the log4j2.properties file found in the broker instance etc directory, which is configured by default to log to both the console and to a file.

There are a handful of general loggers available:

<table>
<thead>
<tr>
<th>Logger</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rootLogger</td>
<td>Logs any calls not handled by the Apache ActiveMQ Artemis loggers</td>
</tr>
<tr>
<td>org.apache.activemq.artemis.core.server</td>
<td>Logs the core server</td>
</tr>
<tr>
<td>org.apache.activemq.artemis.utils</td>
<td>Logs utility calls</td>
</tr>
<tr>
<td>org.apache.activemq.artemis.journal</td>
<td>Logs Journal calls</td>
</tr>
<tr>
<td>org.apache.activemq.artemis.jms</td>
<td>Logs JMS calls</td>
</tr>
<tr>
<td>org.apache.activemq.artemis.integration.bootstrap</td>
<td>Logs bootstrap calls</td>
</tr>
<tr>
<td>org.apache.activemq.audit.base</td>
<td>audit log. Disabled by default</td>
</tr>
<tr>
<td>org.apache.activemq.audit.resource</td>
<td>resource audit log. Disabled by default</td>
</tr>
<tr>
<td>org.apache.activemq.audit.message</td>
<td>message audit log. Disabled by default</td>
</tr>
</tbody>
</table>

77.1. Configuring a Specific Level for a Logger

Sometimes it is necessary to get more detailed logs from a particular logger. For example, when you're trying to troubleshoot an issue. Say you needed to get TRACE logging from the logger org.foo.

Then you need to configure the logging level for the org.foo logger to TRACE, e.g.:

```
logger.my_logger_ref.name=org.foo
logger.my_logger_ref.level=TRACE
```

77.2. Configuration Reload

Log4J2 has its own configuration file reloading mechanism, which is itself configured via the same log4j2.properties configuration file. To enable reload upon configuration updates, set the monitorInterval config property to the interval in seconds that the file should be monitored for updates, e.g.

```
# Monitor config file every 5 seconds for updates
monitorInterval = 5
```
77.3. Logging in a client application

Firstly, if you want to enable logging on the client side you need to include a logging implementation in your application which supports the the SLF4J facade. Taking Log4J2 as an example logging implementation, since it used by the broker, when using Maven your client and logging dependencies might be e.g.:

```xml
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>artemis-jms-client</artifactId>
  <version>2.34.0</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-slf4j2-impl</artifactId>
  <version>2.22.1</version>
</dependency>
```

The Log4J2 configuration can then be supplied via file called `log4j2.properties` on the classpath which will then be picked up automatically.

Alternatively, use of a specific configuration file can be configured via system property `log4j2.configurationFile`, e.g.:

```
-Dlog4j2.configurationFile=file:///path/to/custom-log4j2-config.properties
```

The following is an example `log4j2.properties` for a client application, logging at INFO level to the console and a daily rolling file.

```
# Log4J 2 configuration

# Monitor config file every X seconds for updates
monitorInterval = 5

rootLogger.level = INFO
rootLogger.appenderRef.console.ref = console
rootLogger.appenderRef.log_file.ref = log_file

logger.activemq.name=org.apache.activemq
logger.activemq.level=INFO

# Console appender
appender.console.type=Console
appender.console.name=console
appender.console.layout.type=PatternLayout
appender.console.layout.pattern=%d %-5level [%logger] %msg%n
```
### Log file appender

```java
# Log file appender
appender.log_file.type = RollingFile
appender.log_file.name = log_file
appender.log_file.fileName = log/application.log
appender.log_file.filePattern = log/application.log.%d{yyyy-MM-dd}
appender.log_file.layout.type = PatternLayout
appender.log_file.layout.pattern = %d %-5level [%logger] %msg%n
appender.log_file.policies.type = Policies
appender.log_file.policies.cron.type = CronTriggeringPolicy
appender.log_file.policies.cron.schedule = 0 0 0 * * ?
appender.log_file.policies.cron.evaluateOnStartup = true
```

## 77.4. Configuring Broker Audit Logging

There are 3 audit loggers that can be enabled separately and audit different types of broker events, these are:

1. **base**: This is a highly verbose logger that will capture most events that occur on JMX beans.
2. **resource**: This logs the creation of, updates to, and deletion of resources such as addresses and queues as well as authentication. The main purpose of this is to track console activity and access to the broker.
3. **message**: This logs the production and consumption of messages.

All extra logging will negatively impact performance. Whether or not the performance impact is "too much" will depend on your use-case.

These three audit loggers are disabled by default in the broker `log4j2.properties` configuration file:

```properties
... # Audit loggers: to enable change levels from OFF to INFO
logger.audit_base.name = org.apache.activemq.audit.base
logger.audit_base.level = OFF
logger.audit_base.appenderRef.audit_log_file.ref = audit_log_file
logger.audit_base.additivity = false

logger.audit_resource.name = org.apache.activemq.audit.resource
logger.audit_resource.level = OFF
logger.audit_resource.appenderRef.audit_log_file.ref = audit_log_file
logger.audit_resource.additivity = false

logger.audit_message.name = org.apache.activemq.audit.message
logger.audit_message.level = OFF
logger.audit_message.appenderRef.audit_log_file.ref = audit_log_file
logger.audit_message.additivity = false
...
```

To *enable* the audit log change the level to `INFO`, like this:
The 3 audit loggers can be disable/enabled separately.

Once enabled, all audit records are written into a separate log file (by default `audit.log`).

**77.5. More on Log4J2 configuration:**

For more detail on configuring Log4J 2, see its manual.
Chapter 78. Embedding Apache ActiveMQ Artemis

Apache ActiveMQ Artemis is designed as set of simple Plain Old Java Objects (POJOs). This means Apache ActiveMQ Artemis can be instantiated and run in any dependency injection framework such as Spring or Google Guice. It also means that if you have an application that could use messaging functionality internally then it can directly instantiate Apache ActiveMQ Artemis clients and servers in its own application code to perform that functionality. We call this embedding Apache ActiveMQ Artemis.

Examples of applications that might want to do this include any application that needs very high performance, transactional, persistent messaging but doesn't want the hassle of writing it all from scratch.

Embedding Apache ActiveMQ Artemis can be done in very few easy steps - supply a broker.xml on the classpath or instantiate the configuration object, instantiate the server, start it, and you have a Apache ActiveMQ Artemis running in your JVM. It's as simple and easy as that.

78.1. Embedding with XML configuration

The simplest way to embed Apache ActiveMQ Artemis is to use the embedded wrapper class and configure Apache ActiveMQ Artemis through broker.xml.

Here's a simple example broker.xml:

```xml
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="urn:activemq" xsi:schemaLocation="urn:activemq /schema/artemis-server.xsd">
    <core xmlns="urn:activemq:core">
        <persistence-enabled>false</persistence-enabled>
        <security-enabled>false</security-enabled>
        <acceptors>
            <acceptor name="in-vm">vm://0</acceptor>
        </acceptors>
    </core>
</configuration>
```

```java
import org.apache.activemq.artemis.core.server.embedded.EmbeddedActiveMQ;

... 
EmbeddedActiveMQ embedded = new EmbeddedActiveMQ();
embedded.start();
```
ServerLocator serverLocator = ActiveMQClient.createServerLocator("vm://0");
ClientSessionFactory factory = serverLocator.createSessionFactory();
ClientSession session = factory.createSession();

session.createServerLocator("example");

ClientProducer producer = session.createProducer("example");
ClientMessage message = session.createMessage(true);
message.getBody().writeString("Hello");
producer.send(message);

session.start();
ClientConsumer consumer = session.createConsumer("example");
ClientMessage msgReceived = consumer.receive();
System.out.println("message = " + msgReceived.getBody().readString());
session.close();

The EmbeddedActiveMQ class has a few additional setter methods that allow you to specify a different config file name as well as other properties. See the javadocs for this class for more details.

78.2. Embedding with programmatic configuration

You can follow this step-by-step guide to programmatically embed a broker instance.

Create the Configuration object. This contains configuration information for an Apache ActiveMQ Artemis instance. The setter methods of this class allow you to programatically set configuration options as described in the Server Configuration section.

The acceptors are configured through Configuration. Just add the acceptor URL the same way you would through the main configuration file.

```java
import org.apache.activemq.artemis.core.config.Configuration;
import org.apache.activemq.artemis.core.config.impl.ConfigurationImpl;

...

Configuration config = new ConfigurationImpl();
config.addAccepterConfiguration("in-vm", "vm://0");
config.addAccepterConfiguration("tcp", "tcp://127.0.0.1:61616");
```

You need to instantiate an instance of org.apache.activemq.artemis.api.core.server.embedded.EmbeddedActiveMQ and add the configuration object to it.

```java
import org.apache.activemq.artemis.api.core.server.ActiveMQ;
import org.apache.activemq.artemis.core.server.embedded.EmbeddedActiveMQ;
```
... 

```java
EmbeddedActiveMQ server = new EmbeddedActiveMQ();
server.setConfiguration(config);
server.start();
```

You also have the option of instantiating `ActiveMQServerImpl` directly:

```java
ActiveMQServer server = new ActiveMQServerImpl(config);
server.start();
```
Chapter 79. Artemis on Apache Karaf

Apache ActiveMQ Artemis is OSGi ready. Below you can find instruction on how to install and configure broker on Apache Karaf OSGi container.

79.1. Installation

Apache ActiveMQ Artemis provides features that makes it easy to install the broker on Apache Karaf (4.x or later). First you need to define the feature URL, like

```
karaf@root()> feature:repo-add mvn:org.apache.activemq/artemis-features/1.3.0-SNAPSHOT/xml/features
```

This will add Artemis related features

```
karaf@root()> feature:list | grep artemis
artemis              | 1.3.0.SNAPSHOT | | Uninstalled | artemis-
1.3.0-SNAPSHOT       | Full ActiveMQ Artemis broker with default configuration
netty-core            | 4.0.32.Final   | | Uninstalled | artemis-
1.3.0-SNAPSHOT       | Netty libraries
artemis-core          | 1.3.0.SNAPSHOT | | Uninstalled | artemis-
1.3.0-SNAPSHOT       | ActiveMQ Artemis broker libraries
artemis-amqp          | 1.3.0.SNAPSHOT | | Uninstalled | artemis-
1.3.0-SNAPSHOT       | ActiveMQ Artemis AMQP protocol libraries
artemis-stomp         | 1.3.0.SNAPSHOT | | Uninstalled | artemis-
1.3.0-SNAPSHOT       | ActiveMQ Artemis Stomp protocol libraries
artemis-mqtt          | 1.3.0.SNAPSHOT | | Uninstalled | artemis-
1.3.0-SNAPSHOT       | ActiveMQ Artemis MQTT protocol libraries
artemis-hornetq       | 1.3.0.SNAPSHOT | | Uninstalled | artemis-
1.3.0-SNAPSHOT       | ActiveMQ Artemis HornetQ protocol libraries
```

Feature named **artemis** contains full broker installation, so running

```
feature:install artemis
```

will install and run the broker.

79.2. Configuration

The broker is installed as **org.apache.activemq.artemis** OSGi component, so it’s configured through `${KARAF_BASE}/etc/org.apache.activemq.artemis.cfg` file. An example of the file looks like

```
config=file:etc/artemis.xml
name=local
domain=karaf
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td>Location of the configuration file</td>
<td><code>${KARAF_BASE}/etc/artemis.xml</code></td>
</tr>
<tr>
<td>name</td>
<td>Name of the broker</td>
<td>local</td>
</tr>
<tr>
<td>domain</td>
<td>JAAS domain to use for security</td>
<td>karaf</td>
</tr>
<tr>
<td>rolePrincipalClass</td>
<td>Class name used for role authorization purposes</td>
<td>org.apache.karaf.jaas.boot.principal.RolePrincipal</td>
</tr>
</tbody>
</table>

The default broker configuration file is located in `${KARAF_BASE}/etc/artemis.xml`
Chapter 80. Apache Tomcat Support

80.1. Resource Context Client Configuration

Apache ActiveMQ Artemis provides support for configuring the client in the context.xml of the Tomcat container.

This is very similar to the way this is done in ActiveMQ "Classic" so anyone migrating should find this familiar. Please note though the connection url and properties that can be set for ActiveMQ Artemis are different please see Migration Documentation

80.1.1. Example of Connection Factory

```
<Context>
  ...
  <Resource name="jms/ConnectionFactory" auth="Container"
      type="org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory"
      description="JMS Connection Factory"
      factory="org.apache.activemq.artemis.jndi.JNDIReferenceFactory"
      brokerURL="tcp://localhost:61616" />
  ...
</Context>
```

80.1.2. Example of Destination (Queue and Topic)

```
<Context>
  ...
  <Resource name="jms/ExampleQueue" auth="Container"
      type="org.apache.activemq.artemis.jms.client.ActiveMQQueue"
      description="JMS Queue"
      factory="org.apache.activemq.artemis.jndi.JNDIReferenceFactory"
      address="ExampleQueue" />
  ...
  <Resource name="jms/ExampleTopic" auth="Container"
      type="org.apache.activemq.artemis.jms.client.ActiveMQTopic"
      description="JMS Topic"
      factory="org.apache.activemq.artemis.jndi.JNDIReferenceFactory"
      address="ExampleTopic" />
  ...
</Context>
```

80.2. Example Tomcat App

A sample Tomcat app with the container context configured as an example can be seen here:

/examples/features/sub-modules/tomcat
Chapter 81. CDI Integration

Apache ActiveMQ Artemis provides a simple CDI integration. It can either use an embedded broker or connect to a remote broker.

81.1. Configuring a connection

Configuration is provided by implementing the `ArtemisClientConfiguration` interface.

```java
public interface ArtemisClientConfiguration {
    String getHost();

    Integer getPort();

    String getUsername();

    String getPassword();

    String getUrl();

    String getConnectorFactory();

    boolean startEmbeddedBroker();

    boolean isHa();

    boolean hasAuthentication();
}
```

There’s a default configuration out of the box, if none is specified. This will generate an embedded broker.
Chapter 82. Properties for Copied Messages

There are several operations within the broker that result in copying a message. These include:

- Diverting a message from one address to another.
- Moving an expired message from a queue to the configured `expiry-address`
- Moving a message which has exceeded its `max-delivery-attempts` from a queue to the configured `dead-letter-address`
- Using the management API to administratively move messages from one queue to another

When this happens the body and properties of the original message are copied to a new message. However, the copying process removes some potentially important pieces of data so those are preserved in the following special message properties:

- **_AMQ_ORIG_ADDRESS**
  - a String property containing the `original address` of the message
- **_AMQ_ORIG_QUEUE**
  - a String property containing the `original queue` of the message
- **_AMQ_ORIG_MESSAGE_ID**
  - a String property containing the `original message ID` of the message

It's possible for the aforementioned operations to be combined. For example, a message may be diverted from one address to another where it lands in a queue and a consumer tries & fails to consume it such that the message is then sent to a dead-letter address. Or a message may be administratively moved from one queue to another where it then expires.

In cases like these the `ORIG` properties will contain the information from the last (i.e. most recent) operation.
Chapter 83. Maven Plugins

Since Artemis 1.1.0 Artemis provides the possibility of using Maven Plugins to manage the life cycle of servers.

83.1. When to use it

These Maven plugins were initially created to manage server instances across our examples. They can create a server, start, and do any CLI operation over servers.

You could for example use these maven plugins on your testsuite or deployment automation.

83.2. Goals

There are three goals that you can use

create

This will create a server accordingly to your arguments. You can do some extra tricks here such as installing extra libraries for external modules.

cli

This will perform any CLI operation. This is basically a maven expression of the CLI classes

runClient

This is a simple wrapper around classes implementing a static main call. Notice that this won’t spawn a new VM or new Thread.

83.3. Declaration

On your pom, use the plugins section:

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.activemq</groupId>
      <artifactId>artemis-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

83.4. create goal

I won’t detail every operation of the create plugin here, but I will try to describe the main parameters:
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>configuration</td>
<td>A place that will hold any file to replace on the configuration. For instance if you are providing your own broker.xml. Default is &quot;${basedir}/target/classes/activemq/server0&quot;</td>
</tr>
<tr>
<td>home</td>
<td>The location where you downloaded and installed artemis. Default is &quot;${activemq.basedir}&quot;</td>
</tr>
<tr>
<td>alternateHome</td>
<td>This is used case you have two possible locations for your home (e.g. one under compile and one under production)</td>
</tr>
<tr>
<td>instance</td>
<td>Where the server is going to be installed. Default is &quot;${basedir}/target/server0&quot;</td>
</tr>
<tr>
<td>liblist[]</td>
<td>A list of libraries to be installed under ./lib. ex: &quot;org.jgroups:jgroups:3.6.0.Final&quot;</td>
</tr>
</tbody>
</table>

Example:

```xml
<execution>
  <id>create</id>
  <goals>
    <goal>create</goal>
  </goals>
  <configuration>
    <ignore>${noServer}</ignore>
  </configuration>
</execution>
```

### 83.5. cli goal

Some properties for the CLI

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>configuration</td>
<td>A place that will hold any file to replace on the configuration. For instance if you are providing your own broker.xml. Default is &quot;${basedir}/target/classes/activemq/server0&quot;</td>
</tr>
<tr>
<td>home</td>
<td>The location where you downloaded and installed artemis. Default is &quot;${activemq.basedir}&quot;</td>
</tr>
<tr>
<td>alternateHome</td>
<td>This is used case you have two possible locations for your home (e.g. one under compile and one under production)</td>
</tr>
</tbody>
</table>
Similarly to the create plugin, the artemis examples are using the cli plugin. Look at them for concrete examples.

**Example:**

```xml
<execution>
  <id>start</id>
  <goals>
    <goal>cli</goal>
  </goals>
  <configuration>
    <spawn>true</spawn>
    <ignore>${noServer}</ignore>
    <testURI>tcp://localhost:61616</testURI>
    <args>
      <param>run</param>
    </args>
  </configuration>
</execution>
```

### 83.5.1. runClient goal

This is a simple solution for running classes implementing the main method.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clientClass</td>
<td>A class implement a static void main(String arg[])</td>
</tr>
<tr>
<td>args</td>
<td>A string array of arguments passed to the method</td>
</tr>
</tbody>
</table>

**Example:**

```xml
<execution>
  <id>runClient</id>
  <goals>
    <goal>runClient</goal>
  </goals>
  <configuration>
    <clientClass>org.apache.activemq.artemis.jms.example.QueueExample</clientClass>
  </configuration>
</execution>
```
83.5.2. Complete example

The following example is a copy of the /examples/features/standard/queue example. You may refer to it directly under the examples directory tree.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.apache.activemq.examples.broker</groupId>
        <artifactId>jms-examples</artifactId>
        <version>1.1.0</version>
    </parent>
    <artifactId>queue</artifactId>
    <packaging>jar</packaging>
    <name>ActiveMQ Artemis JMS Queue Example</name>
    <properties>
        <activemq.basedir>${project.basedir}/../../../..
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.apache.activemq</groupId>
            <artifactId>artemis-jms-client</artifactId>
            <version>${project.version}</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.activemq</groupId>
                <artifactId>artemis-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>create</id>
                        <goals>
                            <goal>create</goal>
                        </goals>
                        <configuration>
                            <ignore>${noServer}</ignore>
                        </configuration>
                    </execution>
                    <execution>
                        <id>start</id>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```
<goals>
   <goal>cli</goal>
</goals>

<configuration>
   <spawn>true</spawn>
   <ignore>${noServer}</ignore>
   <testURI>tcp://localhost:61616</testURI>
   <args>
      <param>run</param>
   </args>
</configuration>

<configuration>
   <clientClass>
      org.apache.activemq.artemis.jms.example.QueueExample
   </clientClass>
</configuration>

<configuration>
   <ignore>${noServer}</ignore>
</configuration>

<configuration>
   <ignore>${noServer}</ignore>
</configuration>

<dependencies>
   <dependency>
      <groupId>org.apache.activemq.examples.broker</groupId>
      <artifactId>queue</artifactId>
      <version>${project.version}</version>
   </dependency>
</dependencies>

</plugin>

</plugins>

</build>

</project>
Chapter 84. Unit Testing

Artemis resources can be run inside JUnit Tests by using provided Rules (for JUnit 4) or Extensions (for JUnit 5). This can make it easier to embed messaging functionality in your tests.

These are provided by the packages `artemis-junit` (JUnit 4) and `artemis-junit-5` (JUnit 5).

84.1. Examples

84.1.1. JUnit 4

Add Maven dependency

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>artemis-junit</artifactId>
  <!-- replace this for the version you are using ⟷
  <version>2.34.0</version>
  <scope>test</scope>
</dependency>
```

Declare a rule on your JUnit Test

```
import org.apache.activemq.artemis.junit.EmbeddedActiveMQResource;
import org.junit.Rule;
import org.junit.Test;

public class MyTest {

  @Rule
  public EmbeddedActiveMQResource server = new EmbeddedActiveMQResource();

  @Test
  public void myTest() {
    // test something, eg. create a queue
    server.createQueue("test.adress", "test.queue");
  }
}
```

84.1.2. JUnit 5

Add Maven dependency

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>artemis-junit-5</artifactId>
</dependency>
```
Declare a rule on your JUnit Test

```java
import org.apache.activemq.artemis.junit.EmbeddedActiveMQExtension;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;

public class MyTest {
    @RegisterExtension
    public EmbeddedActiveMQExtension server = new EmbeddedActiveMQExtension();
    
    @Test
    public void myTest() {
        // test something, eg. create a queue
        server.createQueue("test.adress", "test.queue");
    }
}
```

84.2. Ordering rules / extensions

This is actually a JUnit feature, but this could be helpful on pre-determining the order on which rules are executed.

84.2.1. JUnit 4

```java
import org.junit.Rule;
import org.junit.rules.RuleChain;

public EmbeddedActiveMQResource server = new EmbeddedActiveMQResource();
public ActiveMQDynamicProducerResource producer = new ActiveMQDynamicProducerResource(server.getVmURL());

@Rule
public RuleChain ruleChain = RuleChain.outerRule(server).around(producer);
```

84.2.2. JUnit 5

```java
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.extension.RegisterExtension;

@RegisterExtension
```
```java
@Order(1)
public EmbeddedActiveMQExtension producer = new EmbeddedActiveMQExtension();

@RegisterExtension
@Order(2)
public ActiveMQDynamicProducerExtension producer = new ActiveMQDynamicProducerExtension(server.getVmURL());
```

84.3. Available Rules / Extensions

<table>
<thead>
<tr>
<th>JUnit 4 Rule</th>
<th>JUnit 5 Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmbeddedActiveMQResource</td>
<td>EmbeddedActiveMQExtension</td>
<td>Run a Server, without the JMS manager</td>
</tr>
<tr>
<td>EmbeddedJMSResource (deprecated)</td>
<td>no equivalent</td>
<td>Run a Server, including the JMS Manager</td>
</tr>
<tr>
<td>ActiveMQConsumerResource</td>
<td>ActiveMQConsumerExtension</td>
<td>Automate the creation of a consumer</td>
</tr>
<tr>
<td>ActiveMQDynamicProducerResource</td>
<td>ActiveMQDynamicProducerExtension</td>
<td>Automate the creation of a producer</td>
</tr>
<tr>
<td>ActiveMQProducerResource</td>
<td>ActiveMQProducerExtension</td>
<td>Automate the creation of a producer</td>
</tr>
</tbody>
</table>
Chapter 85. JCA Resource Adapter

For using Apache ActiveMQ Artemis in a Java EE or Jakarta EE environment, you can use the JCA Resource Adapter.

A JCA-based JMS connection factory has 2 big advantages over a plain JMS connection factory:

- **Pooled** - Generally speaking, when a connection is "created" from a JCA-based JMS connection factory the underlying physical connection is taken out of a pool and when the connection is "closed" the underlying physical connection is returned to the pool. This eliminates the performance penalty of actually creating and destroying the physical connection which allows clients to be written in ways that would normally be considered an anti-pattern (e.g. "creating" and "closing" a connection for every sent message).

- **Automatic enlistment into JTA transactions** - Most of the time applications which consume JMS messages in a Java/Jakarta EE context do so via an MDB. By default, the consumption of the message in an MDB (i.e. the execution of `onMessage`) happens within a JTA transaction. If a JCA-based JMS connection factory is used in the course of the MDB's processing (e.g. to send a message) then the JCA logic will automatically enlist the session into the JTA transaction so that the consumption of the message and the sending of the message are an atomic operation (assuming that the JCA-based connection factory is XA capable). This is also true for operations involving other transactional resources (e.g. a database).

### 85.1. Versions

Pick the right version of the resource adapter depending on your environment.

<table>
<thead>
<tr>
<th></th>
<th>artemis-ra-rar</th>
<th></th>
<th>artemis-jakarta-ra-rar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java EE</td>
<td>JCA</td>
<td>JMS</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.7</td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>

### 85.2. Building the RA

To use the RA you have to build it. The simplest way to do this is with the examples.

```
  cd examples/features/sub-modules/{artemis-jakarta-ra-rar,artemis-ra-rar}
mvn clean install
cd target
mv artemis*.rar artemis.rar
```

Follow the manual of your application server to install the artemis.rar JCA RA archive.
85.3. Configuration

The configuration is split into two parts. First the config to send messages to a destination (outbound), and second the config to get messages consumed from a destination (inbound). Each can be configured separately or both can use the ResourceAdapter settings.

Here are a few options listed. If you want an overview of all configuration options, consider https://github.com/apache/activemq-artemis/blob/{ config.version }/artemis-ra/src/main/java/org/apache/activemq/artemis/ra/ConnectionFactoryProperties.java[ConnectionFactoryProperties] as a base and additionally the specific classes for your object.

Consider also the rar.xml file for options and explanations in your artemis.rar. There you can set the default options for your ResourceAdapter. With the configuration of the ResourceAdapter in your application server, you are overriding rar.xml defaults. With the configuration of the ConnectionFactory or the ActivationSpec, you can override the ResourceAdapter config.

85.3.1. ResourceAdapter


connectionParameters
  key value pairs, like host=localhost;port=61616,host=anotherHost;port=61617

userName
  userName

password
  password

clientID
  clientID

85.3.2. ConnectionFactory

Config options for the outbound ManagedConnectionFactory: https://github.com/apache/activemq-artemis/blob/{ config.version }/artemis-ra/src/main/java/org/apache/activemq/artemis/ra/ActiveMQRAMCFProperties.java[ActiveMQRAMCFProperties] The connection for the ManagedConnectionFactory is specified by the RA.


brokerUrl
  url to broker
**cacheDestinations**
by the jms session

**ConnectionManager**
You can't configure any properties.

### 85.3.3. ActivationSpec


In the activation spec you can configure all the things you need to get messages consumed from ActiveMQ Artemis.

**useJndi**
true if you want lookup destinations via jndi.

**connectionFactoryLookup**
The jndiName of the connectionFactory, used by this activation spec. You can reference an existing ManagedConnectionFactory or specify another.

**jndiParams**
For the InitialContext. key value pairs, like `a=b;c=d;e=f`

**destination**
name or JNDI reference of the JMS destination

**destinationType**
[javax|jakarta].jms.Queue or [javax|jakarta].jms.Topic

**messageSelector**
JMS selector to filter messages to your MDB

**maxSession**
to consume messages in parallel from the broker

**Only for topic message consumption**

**subscriptionDurability**
Durable / NonDurable

**subscriptionName**
Artemis holds all messages for this name if you use durable subscriptions
85.4. Logging

With the package `org.apache.activemq.artemis.ra` you catch all ResourceAdapter logging statements.
Chapter 86. Configuration Reference

This section is a quick index for looking up configuration. Click on the element name to go to the specific chapter.

86.1. Broker Configuration

86.1.1. broker.xml

This is the main core server configuration file which contains the core element. The core element contains the main server configuration.

Modularising broker.xml

XML XInclude support is provided in broker.xml so that you can break your configuration out into separate files.

To do this ensure the following is defined at the root configuration element.

<xi:include href="my-address-settings.xml"/>

You should ensure xml elements in separated files should be namespaced correctly for example if address-settings element was separated, it should have the element namespace defined:

<address-settings xmlns="urn:activemq:core">

An example can of this feature can be seen in the test suites:

./artemis-server/src/test/resources/ConfigurationTest-xinclude-config.xml

if you use xmllint to validate the XML against the schema you should enable xinclude flag when running.

--xinclude

For further information on XInclude see: https://www.w3.org/TR/xinclude/

To disable XML external entity processing use the system property artemis.disableXxe, e.g.: 
Reloading modular configuration files

Certain changes in `broker.xml` can be picked up at runtime as discussed in the Configuration Reload chapter. Changes made directly to files which are included in `broker.xml` via `xi:include` will not be automatically reloaded. For example, if `broker.xml` is including `my-address-settings.xml` and `my-address-settings.xml` is modified those changes won’t be reloaded automatically. To force a reload in this situation there are 2 main options:

1. Use the `reloadConfiguration` management operation on the `ActiveMQServerControl`.
2. Update the timestamp on `broker.xml` using something like the `touch` command. The next time the broker inspects `broker.xml` for automatic reload it will see the updated timestamp and trigger a reload of `broker.xml` and all its included files.

86.1.2. System properties

It is possible to use System properties to replace some of the configuration properties. If you define a System property starting with "brokerconfig." that will be passed along to Bean Utils and the configuration would be replaced.

To define `global-max-size=1000000` using a system property you would have to define this property, for example through java arguments:

```
java -Dbrokerconfig.globalMaxSize=1000000
```

You can also change the prefix through the `broker.xml` by setting:

```
<system-property-prefix>yourprefix</system-property-prefix>
```

This is to help you customize artemis on embedded systems.

86.1.3. Broker properties

Broker properties extends the use of properties to allow updates and additions to the broker configuration after any xml has been parsed. In the absence of any broker.xml, the hard coded defaults can be modified. Internally, any xml configuration is applied to a java bean style configuration object. Typically, there are setters for each of the xml attributes. However, for properties, the naming convention changes from ‘a-b’ to ‘aB’ to reflect the camelCase java naming convention.

Collections need some special treatment to allow additions and reference. We utilise the name attribute of configuration entities to find existing entries and when populating new entities, we set the name to match the requested key. Removal of configuration from named collections is supported by setting a key value to "-". The remove match value can be configured with a property key "remove.value".
For example, a properties file containing:

```
securityEnabled=false
acceptorConfigurations.tcp.factoryClassName=org.apache.activemq.artemis.core.remoting.impl.netty.NettyAcceptorFactory
acceptorConfigurations.tcp.params.HOST=localhost
acceptorConfigurations.tcp.params.PORT=61616
```

would: 1) disable RBAC security checks 2) add or modify an acceptor named "tcp" that will use Netty 3) set the acceptor named "tcp" 'HOST' parameter to localhost 4) set the acceptor named "tcp" 'PORT' parameter to 61616

The configuration properties are low level, lower level than xml, however it is very powerful; any accessible attribute of the internal `org.apache.activemq.artemis.core.config.impl.ConfigurationImpl` objects can be modified.

With great power one must take great care!

The `artemis run` command script supports `--properties <properties file url>`, where a properties file can be configured.

One shortcoming of this method of configuration is that any property that does not match is ignored with no fanfare. Enable debug logging for `org.apache.activemq.artemis.core.config.impl.ConfigurationImpl` to get more insight.

There are a growing number of examples of what can be explicitly configured in this way in the unit test.

### 86.2. The core configuration

This describes the root of the XML configuration. You will see here also multiple sub-types listed. For example on the main config you will have bridges and at the list of bridge type we will describe the properties for that configuration.

The default values listed below are the values which will be used if the configuration parameter is **not set** either programmatically or via `broker.xml`. Some of these values are set in the `broker.xml` which is available out-of-the-box. Any values set in the out-of-the-box configuration will override the default values listed here. Please consult your specific configuration to know which values will actually be used when the broker is running.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>acceptors</td>
<td>a list of remoting acceptors</td>
<td>n/a</td>
</tr>
<tr>
<td>acceptors.acceptor</td>
<td>Each acceptor is composed for just an URL</td>
<td>n/a</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>addresses</td>
<td>a list of addresses</td>
<td>n/a</td>
</tr>
<tr>
<td>address-settings</td>
<td>a list of address-setting</td>
<td>n/a</td>
</tr>
<tr>
<td>allow-failback</td>
<td>Should stop backup on primary restart.</td>
<td>true</td>
</tr>
<tr>
<td>amqp-use-core-subscription-naming</td>
<td>If true uses CORE queue naming convention for AMQP.</td>
<td>false</td>
</tr>
<tr>
<td>async-connection-execution-enabled</td>
<td>If False delivery would be always asynchronous.</td>
<td>true</td>
</tr>
<tr>
<td>bindings-directory</td>
<td>The folder in use for the bindings folder</td>
<td>data/bindings</td>
</tr>
<tr>
<td>bridges</td>
<td>a list of core bridges</td>
<td>n/a</td>
</tr>
<tr>
<td>ha-policy</td>
<td>the HA policy of this server</td>
<td>none</td>
</tr>
<tr>
<td>broadcast-groups</td>
<td>a list of broadcast-group</td>
<td>n/a</td>
</tr>
<tr>
<td>broker-connections</td>
<td>a list of amqp-connection</td>
<td>n/a</td>
</tr>
<tr>
<td>broker-plugins</td>
<td>a list of broker-plugins</td>
<td>n/a</td>
</tr>
<tr>
<td>configuration-file-refresh-period</td>
<td>The frequency in milliseconds the configuration file is checked for changes</td>
<td>5000</td>
</tr>
<tr>
<td>check-for-active-server</td>
<td>Used by a primary server to verify if there are other nodes with the same ID on the topology</td>
<td>n/a</td>
</tr>
<tr>
<td>cluster-connections</td>
<td>a list of cluster-connection</td>
<td>n/a</td>
</tr>
<tr>
<td>cluster-password</td>
<td>Cluster password. It applies to all cluster configurations.</td>
<td>n/a</td>
</tr>
<tr>
<td>cluster-user</td>
<td>Cluster username. It applies to all cluster configurations.</td>
<td>n/a</td>
</tr>
<tr>
<td>connection-ttl-override</td>
<td>if set, this will override how long (in ms) to keep a connection alive without receiving a ping. -1 disables this setting.</td>
<td>-1</td>
</tr>
<tr>
<td>connection-ttl-check-interval</td>
<td>how often (in ms) to check connections for ttl violation.</td>
<td>2000</td>
</tr>
<tr>
<td>connectors.connector</td>
<td>The URL for the connector. This is a list</td>
<td>n/a</td>
</tr>
<tr>
<td>create-bindings-dir</td>
<td>true means that the server will create the bindings directory on start up.</td>
<td>true</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>create-journal-dir</td>
<td>true means that the journal directory will be created.</td>
<td>true</td>
</tr>
<tr>
<td>discovery-groups</td>
<td>a list of discovery-group</td>
<td>n/a</td>
</tr>
<tr>
<td>disk-scan-period</td>
<td>The interval where the disk is scanned for percentual usage.</td>
<td>5000</td>
</tr>
<tr>
<td>diverts</td>
<td>a list of diverts to use</td>
<td>n/a</td>
</tr>
<tr>
<td>global-max-size</td>
<td>The amount in bytes before all addresses are considered full.</td>
<td>Half of the JVM’s -Xmx</td>
</tr>
<tr>
<td>graceful-shutdown-enabled</td>
<td>true means that graceful shutdown is enabled.</td>
<td>false</td>
</tr>
<tr>
<td>graceful-shutdown-timeout</td>
<td>Timeout on waiting for clients to disconnect before server shutdown.</td>
<td>-1</td>
</tr>
<tr>
<td>grouping-handler</td>
<td>a message grouping handler</td>
<td>n/a</td>
</tr>
<tr>
<td>id-cache-size</td>
<td>The duplicate detection circular cache size.</td>
<td>20000</td>
</tr>
<tr>
<td>jmx-domain</td>
<td>the JMX domain used to registered MBeans in the MBeanServer.</td>
<td>org.apache.activemq</td>
</tr>
<tr>
<td>jmx-use-broker-name</td>
<td>whether or not to use the broker name in the JMX properties.</td>
<td>true</td>
</tr>
<tr>
<td>jmx-management-enabled</td>
<td>true means that the management API is available via JMX.</td>
<td>true</td>
</tr>
<tr>
<td>journal-buffer-size</td>
<td>The size of the internal buffer on the journal in KB.</td>
<td>490KB</td>
</tr>
<tr>
<td>journal-buffer-timeout</td>
<td>The Flush timeout for the journal buffer</td>
<td>500000 for ASYNCIO; 3333333 for NIO</td>
</tr>
<tr>
<td>journal-compact-min-files</td>
<td>The minimal number of data files before we can start compacting. Setting this to 0 means compacting is disabled.</td>
<td>10</td>
</tr>
<tr>
<td>journal-compact-percentage</td>
<td>The percentage of live data on which we consider compacting the journal.</td>
<td>30</td>
</tr>
<tr>
<td>journal-directory</td>
<td>the directory to store the journal files in.</td>
<td>data/journal</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>node-manager-lock-directory</td>
<td>the directory to store the node manager lock file.</td>
<td>same of journal-directory</td>
</tr>
<tr>
<td>journal-file-size</td>
<td>the size (in bytes) of each journal file.</td>
<td>10MB</td>
</tr>
<tr>
<td>journal-lock-acquisition-timeout</td>
<td>how long (in ms) to wait to acquire a file lock on the journal.</td>
<td>-1</td>
</tr>
<tr>
<td>journal-max-io</td>
<td>the maximum number of write requests that can be in the ASYNCIO queue at any one time.</td>
<td>4096 for ASYNCIO; 1 for NIO; ignored for MAPPED</td>
</tr>
<tr>
<td>journal-file-open-timeout</td>
<td>the length of time in seconds to wait when opening a new journal file before timing out and failing.</td>
<td>5</td>
</tr>
<tr>
<td>journal-min-files</td>
<td>how many journal files to pre-create.</td>
<td>2</td>
</tr>
<tr>
<td>journal-pool-files</td>
<td>The upper threshold of the journal file pool, -1 means no Limit. The system will create as many files as needed however when reclaiming files it will shrink back to the journal-pool-files</td>
<td>-1</td>
</tr>
<tr>
<td>journal-sync-non-transactional</td>
<td>if true wait for non transaction data to be synced to the journal before returning response to client.</td>
<td>true</td>
</tr>
<tr>
<td>journal-sync-transactional</td>
<td>if true wait for transaction data to be synchronized to the journal before returning response to client.</td>
<td>true</td>
</tr>
<tr>
<td>journal-type</td>
<td>the type of journal to use.</td>
<td>ASYNCIO</td>
</tr>
<tr>
<td>journal-datasync</td>
<td>It will use msync/fsync on journal operations.</td>
<td>true</td>
</tr>
<tr>
<td>large-messages-directory</td>
<td>the directory to store large messages.</td>
<td>data/largemessages</td>
</tr>
<tr>
<td>log-delegate-factory-class-name</td>
<td>deprecated the name of the factory class to use for log delegation.</td>
<td>n/a</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>management-address</td>
<td>the name of the management address to send management messages to.</td>
<td>activemq.management</td>
</tr>
<tr>
<td>management-notification-address</td>
<td>the name of the address that consumers bind to receive management notifications.</td>
<td>activemq.notifications</td>
</tr>
<tr>
<td>mask-password</td>
<td>This option controls whether passwords in server configuration need be masked. If set to &quot;true&quot; the passwords are masked.</td>
<td>false</td>
</tr>
<tr>
<td>max-saved-replicated-journals-size</td>
<td>This specifies how many replication backup directories will be kept when server starts as replica. -1 Means no Limit; 0 don’t keep a copy at all.</td>
<td>2</td>
</tr>
<tr>
<td>max-disk-usage</td>
<td>The max percentage of data we should use from disks. The broker will block while the disk is full. Disable by setting -1.</td>
<td>90</td>
</tr>
<tr>
<td>memory-measure-interval</td>
<td>frequency to sample JVM memory in ms (or -1 to disable memory sampling).</td>
<td>-1</td>
</tr>
<tr>
<td>memory-warning-threshold</td>
<td>Percentage of available memory which will trigger a warning log.</td>
<td>25</td>
</tr>
<tr>
<td>message-counter-enabled</td>
<td>true means that message counters are enabled.</td>
<td>false</td>
</tr>
<tr>
<td>message-counter-max-day-history</td>
<td>how many days to keep message counter history.</td>
<td>10</td>
</tr>
<tr>
<td>message-counter-sample-period</td>
<td>the sample period (in ms) to use for message counters.</td>
<td>10000</td>
</tr>
<tr>
<td>message-expiry-scan-period</td>
<td>how often (in ms) to scan for expired messages.</td>
<td>30000</td>
</tr>
<tr>
<td>message-expiry-thread-priority</td>
<td>deprecated the priority of the thread expiring messages.</td>
<td>3</td>
</tr>
<tr>
<td>metrics-plugin</td>
<td>a plugin to export metrics</td>
<td>n/a</td>
</tr>
<tr>
<td>address-queue-scan-period</td>
<td>how often (in ms) to scan for addresses &amp; queues that should be removed.</td>
<td>30000</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>name</td>
<td>node name; used in topology notifications if set.</td>
<td>n/a</td>
</tr>
<tr>
<td>password-codec</td>
<td>the name of the class (and optional configuration properties) used to decode masked passwords. Only valid when mask-password is true.</td>
<td>n/a</td>
</tr>
<tr>
<td>page-max-concurrent-io</td>
<td>The max number of concurrent reads allowed on paging.</td>
<td>5</td>
</tr>
<tr>
<td>page-sync-timeout</td>
<td>The time in nanoseconds a page will be synced.</td>
<td>3333333 for ASYNCIO; journal-buffer-timeout for NIO</td>
</tr>
<tr>
<td>read-whole-page</td>
<td>If true the whole page would be read, otherwise just seek and read while getting message.</td>
<td>false</td>
</tr>
<tr>
<td>paging-directory</td>
<td>the directory to store paged messages in.</td>
<td>data/paging</td>
</tr>
<tr>
<td>persist-delivery-count-before-delivery</td>
<td>True means that the delivery count is persisted before delivery. False means that this only happens after a message has been cancelled.</td>
<td>false</td>
</tr>
<tr>
<td>max-redelivery-records</td>
<td>Maximum number of records the system will store for redeliveries. In most cases this should be set to ‘1’.</td>
<td>10</td>
</tr>
<tr>
<td>persistence-enabled</td>
<td>true means that the server will use the file based journal for persistence.</td>
<td>true</td>
</tr>
<tr>
<td>persist-id-cache</td>
<td>true means that ID's are persisted to the journal.</td>
<td>true</td>
</tr>
<tr>
<td>queues</td>
<td><strong>deprecated</strong> use addresses</td>
<td>n/a</td>
</tr>
<tr>
<td>remoting-incoming-interceptors</td>
<td>a list of <code>&lt;class-name/&gt;</code> elements with the names of classes to use for intercepting incoming remoting packets</td>
<td>n/a</td>
</tr>
<tr>
<td>remoting-outgoing-interceptors</td>
<td>a list of <code>&lt;class-name/&gt;</code> elements with the names of classes to use for intercepting outgoing remoting packets</td>
<td>n/a</td>
</tr>
<tr>
<td>resolveProtocols</td>
<td>Use ServiceLoader to load protocol modules.</td>
<td>true</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>resource-limit-settings</td>
<td>a list of resource-limits</td>
<td>n/a</td>
</tr>
<tr>
<td>scheduled-thread-pool-max-size</td>
<td>Maximum number of threads to use for the scheduled thread pool.</td>
<td>5</td>
</tr>
<tr>
<td>security-enabled</td>
<td>true means that security is enabled.</td>
<td>true</td>
</tr>
<tr>
<td>security-invalidation-interval</td>
<td>how long (in ms) to wait before invalidating the security cache.</td>
<td>10000</td>
</tr>
<tr>
<td>system-property-prefix</td>
<td>Prefix for replacing configuration settings using Bean Utils.</td>
<td>n/a</td>
</tr>
<tr>
<td>internal-naming-prefix</td>
<td>the prefix used when naming the internal queues and addresses required for implementing certain behaviours.</td>
<td>$.activemq.internal</td>
</tr>
<tr>
<td>populate-validated-user</td>
<td>whether or not to add the name of the validated user to the messages that user sends.</td>
<td>false</td>
</tr>
<tr>
<td>security-settings</td>
<td>a list of security-setting.</td>
<td>n/a</td>
</tr>
<tr>
<td>thread-pool-max-size</td>
<td>Maximum number of threads to use for the thread pool. -1 means 'no limits'.</td>
<td>30</td>
</tr>
<tr>
<td>transaction-timeout</td>
<td>how long (in ms) before a transaction can be removed from the resource manager after create time.</td>
<td>300000</td>
</tr>
<tr>
<td>transaction-timeout-scan-period</td>
<td>how often (in ms) to scan for timeout transactions.</td>
<td>1000</td>
</tr>
<tr>
<td>wild-card-routing-enabled</td>
<td>true means that the server supports wild card routing.</td>
<td>true</td>
</tr>
<tr>
<td>network-check-NIC</td>
<td>the NIC (Network Interface Controller) to be used on InetAddress.isReachable.</td>
<td>n/a</td>
</tr>
<tr>
<td>network-check-URL-list</td>
<td>the list of http URIs to be used to validate the network.</td>
<td>n/a</td>
</tr>
<tr>
<td>network-check-list</td>
<td>the list of pings to be used on ping or InetAddress.isReachable.</td>
<td>n/a</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>network-check-period</td>
<td>a frequency in milliseconds to how often we should check if the network is still up.</td>
<td>10000</td>
</tr>
<tr>
<td>network-check-timeout</td>
<td>a timeout used in milliseconds to be used on the ping.</td>
<td>1000</td>
</tr>
<tr>
<td>network-check-ping-command</td>
<td>the command used to ping IPV4 addresses.</td>
<td>n/a</td>
</tr>
<tr>
<td>network-check-ping6-command</td>
<td>the command used to ping IPV6 addresses.</td>
<td>n/a</td>
</tr>
<tr>
<td>critical-analyzer</td>
<td>enable or disable the critical analysis.</td>
<td>true</td>
</tr>
<tr>
<td>critical-analyzer-timeout</td>
<td>timeout used to do the critical analysis.</td>
<td>120000 ms</td>
</tr>
<tr>
<td>critical-analyzer-check-period</td>
<td>time used to check the response times.</td>
<td>0.5 * critical-analyzer-timeout</td>
</tr>
<tr>
<td>critical-analyzer-policy</td>
<td>should the server log, be halted or shutdown upon failures.</td>
<td>LOG</td>
</tr>
<tr>
<td>resolve-protocols</td>
<td>if true then the broker will make use of any protocol managers that are in available on the classpath, otherwise only the core protocol will be available, unless in embedded mode where users can inject their own protocol managers.</td>
<td>true</td>
</tr>
<tr>
<td>resource-limit-settings</td>
<td>a list of resource-limit.</td>
<td>n/a</td>
</tr>
<tr>
<td>server-dump-interval</td>
<td>interval to log server specific information (e.g. memory usage etc).</td>
<td>-1</td>
</tr>
<tr>
<td>store</td>
<td>the store type used by the server.</td>
<td>n/a</td>
</tr>
<tr>
<td>wildcard-addresses</td>
<td>parameters to configure wildcard address matching format.</td>
<td>n/a</td>
</tr>
<tr>
<td>view-permission-method-match-pattern</td>
<td>parameter to configure the regular expression pattern to match management or JMX operations that require the 'view' permission in your security-settings.</td>
<td>^(get</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>management-message-rbac</td>
<td>parameter to enable security-settings RBAC on management messages sent to the management address.</td>
<td>false</td>
</tr>
<tr>
<td>management-rbac-prefix</td>
<td>parameter to configure the prefix for security-settings match addresses to control RBAC on JMX MBean operations and optionally on management messages</td>
<td>mops (shorthand for management operations)</td>
</tr>
</tbody>
</table>

### 86.3. address-setting type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>match</td>
<td>The filter to apply to the setting</td>
<td>n/a</td>
</tr>
<tr>
<td>dead-letter-address</td>
<td>Dead letter address</td>
<td>n/a</td>
</tr>
<tr>
<td>auto-create-dead-letter-resources</td>
<td>Whether or not to auto-create dead-letter address and/or queue</td>
<td>false</td>
</tr>
<tr>
<td>dead-letter-queue-prefix</td>
<td>Prefix to use for auto-created dead-letter queues</td>
<td>DLQ.</td>
</tr>
<tr>
<td>dead-letter-queue-suffix</td>
<td>Suffix to use for auto-created dead-letter queues</td>
<td><code>\</code> (empty)</td>
</tr>
<tr>
<td>expiry-address</td>
<td>Expired messages address</td>
<td>n/a</td>
</tr>
<tr>
<td>expiry-delay</td>
<td>Expiration time override; -1 don’t override</td>
<td>-1</td>
</tr>
<tr>
<td>redelivery-delay</td>
<td>Time to wait before redelivering a message</td>
<td>0</td>
</tr>
<tr>
<td>redelivery-delay-multiplier</td>
<td>Multiplier to apply to the redelivery-delay</td>
<td>1.0</td>
</tr>
<tr>
<td>redelivery-collision-avoidance-factor</td>
<td>an additional factor used to calculate an adjustment to the redelivery-delay (up or down)</td>
<td>0.0</td>
</tr>
<tr>
<td>max-redelivery-delay</td>
<td>Max value for the redelivery-delay</td>
<td>10 * <code>redelivery-delay</code></td>
</tr>
<tr>
<td>max-delivery-attempts</td>
<td>Number of retries before dead letter address</td>
<td>10</td>
</tr>
<tr>
<td>max-size-bytes</td>
<td>Max size a queue can be before invoking <code>address-full-policy</code></td>
<td>-1</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>max-size-bytes-reject-threshold</td>
<td>Used with BLOCK, the max size an address can reach before messages are rejected; works in combination with max-size-bytes for AMQP clients only.</td>
<td>-1</td>
</tr>
<tr>
<td>page-size-bytes</td>
<td>Size of each file on page</td>
<td>10485760</td>
</tr>
<tr>
<td>address-full-policy</td>
<td>What to do when a queue reaches max-size-bytes</td>
<td>PAGE</td>
</tr>
<tr>
<td>message-counter-history-day-limit</td>
<td>Days to keep message counter data</td>
<td>0</td>
</tr>
<tr>
<td>last-value-queue</td>
<td>deprecated Queue is a last value queue; see default-last-value-queue instead</td>
<td>false</td>
</tr>
<tr>
<td>default-last-value-queue</td>
<td>last-value value if none is set on the queue</td>
<td>false</td>
</tr>
<tr>
<td>default-last-value-key</td>
<td>last-value-key value if none is set on the queue</td>
<td>null</td>
</tr>
<tr>
<td>default-exclusive-queue</td>
<td>exclusive value if none is set on the queue</td>
<td>false</td>
</tr>
<tr>
<td>default-non-destructive</td>
<td>non-destructive value if none is set on the queue</td>
<td>false</td>
</tr>
<tr>
<td>default-consumers-before-dispatch</td>
<td>consumers-before-dispatch value if none is set on the queue</td>
<td>0</td>
</tr>
<tr>
<td>default-delay-before-dispatch</td>
<td>delay-before-dispatch value if none is set on the queue</td>
<td>-1</td>
</tr>
<tr>
<td>redistribution-delay</td>
<td>Timeout before redistributing values after no consumers</td>
<td>-1</td>
</tr>
<tr>
<td>send-to-dla-on-no-route</td>
<td>Forward messages to DLA when no queues subscribing</td>
<td>false</td>
</tr>
<tr>
<td>slow-consumer-threshold</td>
<td>Min rate of msgs/sec consumed before a consumer is considered &quot;slow&quot;</td>
<td>-1</td>
</tr>
<tr>
<td>slow-consumer-policy</td>
<td>What to do when &quot;slow&quot; consumer is detected</td>
<td>NOTIFY</td>
</tr>
<tr>
<td>slow-consumer-check-period</td>
<td>How often to check for &quot;slow&quot; consumers</td>
<td>5</td>
</tr>
<tr>
<td>auto-create-jms-queues</td>
<td>deprecated Create JMS queues automatically; see auto-create-queues &amp; auto-create-addresses</td>
<td>true</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>auto-delete-jms-queues</td>
<td><em>deprecated</em> Delete JMS queues automatically; see auto-create-queues &amp; auto-create-addresses</td>
<td>true</td>
</tr>
<tr>
<td>auto-create-jms-topics</td>
<td><em>deprecated</em> Create JMS topics automatically; see auto-create-queues &amp; auto-create-addresses</td>
<td>true</td>
</tr>
<tr>
<td>auto-delete-jms-topics</td>
<td><em>deprecated</em> Delete JMS topics automatically; see auto-create-queues &amp; auto-create-addresses</td>
<td>true</td>
</tr>
<tr>
<td>auto-create-queues</td>
<td>Create queues automatically</td>
<td>true</td>
</tr>
<tr>
<td>auto-delete-queues</td>
<td>Delete auto-created queues automatically</td>
<td>true</td>
</tr>
<tr>
<td>auto-delete-created-queues</td>
<td>Delete created queues automatically</td>
<td>false</td>
</tr>
<tr>
<td>auto-delete-queues-delay</td>
<td>Delay for deleting auto-created queues</td>
<td>0</td>
</tr>
<tr>
<td>auto-delete-queues-message-count</td>
<td>Message count the queue must be at or below before it can be auto deleted</td>
<td>0</td>
</tr>
<tr>
<td>config-delete-queues</td>
<td>How to deal with queues deleted from XML at runtime</td>
<td>OFF</td>
</tr>
<tr>
<td>auto-create-addresses</td>
<td>Create addresses automatically</td>
<td>true</td>
</tr>
<tr>
<td>auto-delete-addresses</td>
<td>Delete auto-created addresses automatically</td>
<td>true</td>
</tr>
<tr>
<td>auto-delete-addresses-delay</td>
<td>Delay for deleting auto-created addresses</td>
<td>0</td>
</tr>
<tr>
<td>config-delete-addresses</td>
<td>How to deal with addresses deleted from XML at runtime</td>
<td>OFF</td>
</tr>
<tr>
<td>config-delete-diverts</td>
<td>How to deal with diverts deleted from XML at runtime</td>
<td>OFF</td>
</tr>
<tr>
<td>management-browse-page-size</td>
<td>Number of messages a management resource can browse</td>
<td>200</td>
</tr>
<tr>
<td>default-purge-on-no-consumers</td>
<td>purge-on-no-consumers value if none is set on the queue</td>
<td>false</td>
</tr>
<tr>
<td>default-max-consumers</td>
<td>max-consumers value if none is set on the queue</td>
<td>-1</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>default-queue-routing-type</td>
<td>Routing type for auto-created queues if the type can't be otherwise determined</td>
<td>MULTICAST</td>
</tr>
<tr>
<td>default-address-routing-type</td>
<td>Routing type for auto-created addresses if the type can't be otherwise determined</td>
<td>MULTICAST</td>
</tr>
<tr>
<td>default-ring-size</td>
<td>The ring-size applied to queues without an explicit ring-size configured</td>
<td>-1</td>
</tr>
<tr>
<td>retroactive-message-count</td>
<td>the number of messages to preserve for future queues created on the matching address</td>
<td>0</td>
</tr>
<tr>
<td>id-cache-size</td>
<td>The duplicate detection circular cache size</td>
<td>Inherits from global id-cache-size</td>
</tr>
</tbody>
</table>

### 86.4. bridge type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>unique name</td>
<td>n/a</td>
</tr>
<tr>
<td>queue-name</td>
<td>name of queue that this bridge consumes from</td>
<td>n/a</td>
</tr>
<tr>
<td>forwarding-address</td>
<td>address to forward to. If omitted original address is used</td>
<td>n/a</td>
</tr>
<tr>
<td>ha</td>
<td>whether this bridge supports fail-over</td>
<td>false</td>
</tr>
<tr>
<td>filter</td>
<td>optional core filter expression</td>
<td>n/a</td>
</tr>
<tr>
<td>transformer-class-name</td>
<td>optional name of transformer class</td>
<td>n/a</td>
</tr>
<tr>
<td>min-large-message-size</td>
<td>Limit before message is considered large.</td>
<td>100KB</td>
</tr>
<tr>
<td>check-period</td>
<td>How often to check for TTL violation. -1 means disabled.</td>
<td>30000</td>
</tr>
<tr>
<td>connection-ttl</td>
<td>TTL for the Bridge. This should be greater than the ping period.</td>
<td>60000</td>
</tr>
<tr>
<td>retry-interval</td>
<td>period (in ms) between successive retries.</td>
<td>2000</td>
</tr>
<tr>
<td>retry-interval-multiplier</td>
<td>multiplier to apply to successive retry intervals.</td>
<td>1</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>max-retry-interval</td>
<td>Limit to the retry-interval growth.</td>
<td>2000</td>
</tr>
<tr>
<td>reconnect-attempts</td>
<td>maximum number of retry attempts.</td>
<td>-1 (no limit)</td>
</tr>
<tr>
<td>use-duplicate-detection</td>
<td>forward duplicate detection headers?</td>
<td>true</td>
</tr>
<tr>
<td>confirmation-window-size</td>
<td>number of bytes before confirmations are sent.</td>
<td>1MB</td>
</tr>
<tr>
<td>producer-window-size</td>
<td>Producer flow control size on the bridge.</td>
<td>-1 (disabled)</td>
</tr>
<tr>
<td>user</td>
<td>Username for the bridge, the default is the cluster username.</td>
<td>n/a</td>
</tr>
<tr>
<td>password</td>
<td>Password for the bridge, default is the cluster password.</td>
<td>n/a</td>
</tr>
<tr>
<td>reconnect-attempts-same-node</td>
<td>Number of retries before trying another node.</td>
<td>10</td>
</tr>
<tr>
<td>routing-type</td>
<td>how to set the routing-type on the bridged message</td>
<td>PASS</td>
</tr>
<tr>
<td>concurrency</td>
<td>Concurrency of the bridge</td>
<td>1</td>
</tr>
</tbody>
</table>

86.5. broadcast-group type

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>unique name</td>
</tr>
<tr>
<td>local-bind-address</td>
<td>Local bind address that the datagram socket is bound to.</td>
</tr>
<tr>
<td>local-bind-port</td>
<td>Local port to which the datagram socket is bound to.</td>
</tr>
<tr>
<td>group-address</td>
<td>Multicast address to which the data will be broadcast.</td>
</tr>
<tr>
<td>group-port</td>
<td>UDP port number used for broadcasting.</td>
</tr>
<tr>
<td>jgroups-file</td>
<td>Name of JGroups configuration file.</td>
</tr>
<tr>
<td>jgroups-channel</td>
<td>Name of JGroups Channel.</td>
</tr>
<tr>
<td>connector-ref</td>
<td>The connector to broadcast.</td>
</tr>
</tbody>
</table>


# 86.6. cluster-connection type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>unique name</td>
<td>n/a</td>
</tr>
<tr>
<td>address</td>
<td>name of the address this cluster connection applies to</td>
<td>n/a</td>
</tr>
<tr>
<td>connector-ref</td>
<td>Name of the connector reference to use.</td>
<td>n/a</td>
</tr>
<tr>
<td>check-period</td>
<td>The period (in milliseconds) used to check if the cluster connection has failed to receive pings from another server</td>
<td>30000</td>
</tr>
<tr>
<td>connection-ttl</td>
<td>Timeout for TTL.</td>
<td>60000</td>
</tr>
<tr>
<td>min-large-message-size</td>
<td>Messages larger than this are considered large-messages.</td>
<td>100KB</td>
</tr>
<tr>
<td>call-timeout</td>
<td>Time(ms) before giving up on blocked calls.</td>
<td>30000</td>
</tr>
<tr>
<td>retry-interval</td>
<td>period (in ms) between successive retries.</td>
<td>500</td>
</tr>
<tr>
<td>retry-interval-multiplier</td>
<td>multiplier to apply to the retry-interval.</td>
<td>1</td>
</tr>
<tr>
<td>max-retry-interval</td>
<td>Maximum value for retry-interval.</td>
<td>2000</td>
</tr>
<tr>
<td>reconnect-attempts</td>
<td>How many attempts should be made to reconnect after failure.</td>
<td>-1</td>
</tr>
<tr>
<td>use-duplicate-detection</td>
<td>should duplicate detection headers be inserted in forwarded messages?</td>
<td>true</td>
</tr>
<tr>
<td>message-load-balancing</td>
<td>how should messages be load balanced?</td>
<td>OFF</td>
</tr>
<tr>
<td>max-hops</td>
<td>maximum number of hops cluster topology is propagated.</td>
<td>1</td>
</tr>
<tr>
<td>confirmation-window-size</td>
<td>The size (in bytes) of the window used for confirming data from the server connected to.</td>
<td>1048576</td>
</tr>
<tr>
<td>producer-window-size</td>
<td>Flow Control for the Cluster connection bridge.</td>
<td>-1 (disabled)</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>call-failover-timeout</td>
<td>How long to wait for a reply if in the middle of a fail-over. -1 means wait forever.</td>
<td>-1</td>
</tr>
<tr>
<td>notification-interval</td>
<td>how often the cluster connection will notify the cluster of its existence right after joining the cluster.</td>
<td>1000</td>
</tr>
<tr>
<td>notification-attempts</td>
<td>how many times this cluster connection will notify the cluster of its existence right after joining the cluster</td>
<td>2</td>
</tr>
</tbody>
</table>

### 86.7. discovery-group type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>unique name</td>
</tr>
<tr>
<td>group-address</td>
<td>Multicast IP address of the group to listen on</td>
</tr>
<tr>
<td>group-port</td>
<td>UDP port number of the multicast group</td>
</tr>
<tr>
<td>jgroups-file</td>
<td>Name of a JGroups configuration file. If specified, the server uses JGroups for discovery.</td>
</tr>
<tr>
<td>jgroups-channel</td>
<td>Name of a JGroups Channel. If specified, the server uses the named channel for discovery.</td>
</tr>
<tr>
<td>refresh-timeout</td>
<td>Period the discovery group waits after receiving the last broadcast from a particular server before removing that server's connector pair entry from its list. Default=10000</td>
</tr>
<tr>
<td>local-bind-address</td>
<td>local bind address that the datagram socket is bound to</td>
</tr>
<tr>
<td>local-bind-port</td>
<td>local port to which the datagram socket is bound to. Default=-1</td>
</tr>
<tr>
<td>initial-wait-timeout</td>
<td>time to wait for an initial broadcast to give us at least one node in the cluster. Default=10000</td>
</tr>
</tbody>
</table>

### 86.8. divert type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>unique name</td>
</tr>
<tr>
<td>transformer-class-name</td>
<td>an optional class name of a transformer</td>
</tr>
<tr>
<td>exclusive</td>
<td>whether this is an exclusive divert. Default=false</td>
</tr>
</tbody>
</table>
### 86.9. address type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>unique name</td>
</tr>
<tr>
<td>anycast</td>
<td>list of anycast queues</td>
</tr>
<tr>
<td>multicast</td>
<td>list of multicast queues</td>
</tr>
</tbody>
</table>

### 86.10. queue type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>unique name</td>
<td>n/a</td>
</tr>
<tr>
<td>filter</td>
<td>optional core filter expression</td>
<td>n/a</td>
</tr>
<tr>
<td>durable</td>
<td>whether the queue is durable (persistent).</td>
<td>true</td>
</tr>
<tr>
<td>user</td>
<td>the name of the user to associate with the creation of the queue</td>
<td>n/a</td>
</tr>
<tr>
<td>max-consumers</td>
<td>the max number of consumers allowed on this queue</td>
<td>-1 (no max)</td>
</tr>
<tr>
<td>purge-on-no-consumers</td>
<td>whether or not to delete all messages and prevent routing when no consumers are connected</td>
<td>false</td>
</tr>
<tr>
<td>exclusive</td>
<td>only deliver messages to one of the connected consumers</td>
<td>false</td>
</tr>
<tr>
<td>last-value</td>
<td>use last-value semantics</td>
<td>false</td>
</tr>
<tr>
<td>ring-size</td>
<td>the size this queue should maintain according to ring semantics</td>
<td>based on default-ring-size address-setting</td>
</tr>
<tr>
<td>consumers-before-dispatch</td>
<td>number of consumers required before dispatching messages</td>
<td>0</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Default</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>delay-before-dispatch</td>
<td>milliseconds to wait for consumers-before-dispatch to be met before dispatching messages anyway</td>
<td>-1 (wait forever)</td>
</tr>
</tbody>
</table>

### 86.11. security-setting type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>match</td>
<td>address expression</td>
</tr>
<tr>
<td>permission</td>
<td></td>
</tr>
<tr>
<td>permission.type</td>
<td>the type of permission</td>
</tr>
<tr>
<td>permission.roles</td>
<td>a comma-separated list of roles to apply the permission to</td>
</tr>
<tr>
<td>role-mapping</td>
<td>A simple role mapping that can be used to map roles from external authentication providers (i.e. LDAP) to internal roles</td>
</tr>
<tr>
<td>role-mapping.from</td>
<td>The external role which should be mapped</td>
</tr>
<tr>
<td>role-mapping.to</td>
<td>The internal role which should be assigned to the authenticated user</td>
</tr>
</tbody>
</table>

### 86.12. broker-plugin type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property</td>
<td>properties to configure a plugin</td>
</tr>
<tr>
<td>class-name</td>
<td>the name of the broker plugin class to instantiate</td>
</tr>
</tbody>
</table>

### 86.13. metrics-plugin type

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property</td>
<td>properties to configure a plugin</td>
</tr>
<tr>
<td>class-name</td>
<td>the name of the metrics plugin class to instantiate</td>
</tr>
</tbody>
</table>

### 86.14. resource-limit type
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>match</td>
<td>the name of the user to whom the limits should be applied</td>
<td>n/a</td>
</tr>
<tr>
<td>max-connections</td>
<td>how many connections are allowed by the matched user</td>
<td>-1 (no max)</td>
</tr>
<tr>
<td>max-queues</td>
<td>how many queues can be created by the matched user</td>
<td>-1 (no max)</td>
</tr>
</tbody>
</table>

**86.15. grouping-handler type**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A unique name</td>
<td>n/a</td>
</tr>
<tr>
<td>type</td>
<td>LOCAL or REMOTE</td>
<td>n/a</td>
</tr>
<tr>
<td>address</td>
<td>A reference to a <code>cluster-connection address</code></td>
<td>n/a</td>
</tr>
<tr>
<td>timeout</td>
<td>How long to wait for a decision</td>
<td>5000</td>
</tr>
<tr>
<td>group-timeout</td>
<td>How long a group binding will be used.</td>
<td>-1 (disabled)</td>
</tr>
<tr>
<td>reaper-period</td>
<td>How often the reaper will be run to check for timed out group bindings. Only valid for LOCAL handlers.</td>
<td>30000</td>
</tr>
</tbody>
</table>

**86.16. amqp-connection type**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri</td>
<td>AMQP broker connection URI (required)</td>
<td>n/a</td>
</tr>
<tr>
<td>name</td>
<td>A unique name</td>
<td>n/a</td>
</tr>
<tr>
<td>user</td>
<td>Broker authentication user (optional)</td>
<td>n/a</td>
</tr>
<tr>
<td>password</td>
<td>Broker authentication password (optional)</td>
<td>n/a</td>
</tr>
<tr>
<td>reconnect-attempts</td>
<td>How many attempts should be made to reconnect after failure.</td>
<td>-1 (infinite)</td>
</tr>
<tr>
<td>auto-start</td>
<td>Broker connection starts automatically with broker</td>
<td>true</td>
</tr>
</tbody>
</table>
Chapter 87. Examples

The Apache ActiveMQ Artemis Examples repository contains over 90 examples demonstrating many of the client and broker features.

The examples can be found at: https://github.com/apache/activemq-artemis-examples

The individual examples are available under the examples directory, and are grouped within the following sub tree:

- **features** - Examples containing broker specific features.
  - **clustered** - examples showing load balancing and distribution capabilities.
  - **ha** - examples showing failover and reconnection capabilities.
  - **perf** - examples allowing you to run a few performance tests on the server
  - **standard** - examples demonstrating various broker features.
  - **sub-modules** - examples of integrated external modules.

- **protocols** - Protocol specific examples
  - **amqp**
  - **mqtt**
  - **openwire**
  - **stomp**

87.1. Running the Examples

First, run **mvn clean package** in the repository root to prepare a broker distribution in the artemis-distribution/target directory for use by the examples.

To run any example, simply **cd** into the appropriate example directory and type **mvn verify** or **mvn install** (For details please read the readme.md in each example directory).

You can use the profile **-Pexamples** to run multiple examples under any example tree.

For each example, you will have a created server under **./target/server0** (some examples use more than one server).

You have the option to prevent the example from starting the server (e.g. if you want to start the server manually) by simply specifying the **-PnoServer** profile, e.g.:

```
# running an example without running the server
mvn verify -PnoServer
```

Also under **./target** there will be a script repeating the commands to create each server. Here is the **create-server0.sh** generated by the **Queue** example. This is useful to see exactly what command(s) are required to configure the server(s).
# These are the commands used to create server0
/myInstallDirectory/apache-artemis/bin/artemis create --allow-anonymous --silent --force --no-web --user guest --password guest --role guest --port-offset 0 --data ./data --allow-anonymous --no-autotune --verbose /myInstallDirectory/apache-artemis-1.1.0/examples/features/standard/queue/target/server0

Several examples use UDP clustering which may not work in your environment by default. On Linux the command would be:

```
route add -net 224.0.0.0 netmask 240.0.0.0 dev lo
```

This command should be run as root. This will redirect any traffic directed to 224.0.0.0 to the loopback interface. On Mac OS X, the command is slightly different:

```
sudo route add 224.0.0.0 127.0.0.1 -netmask 240.0.0.0
```

All the examples use the [Maven plugin](https://mvn.apache.org), which can be useful for running your test servers as well.

This is the common output when running an example. On this case taken from the Queue example:

```
[INFO] --------------< org.apache.activemq.examples.broker:queue >--------------
[INFO] Building ActiveMQ Artemis JMS Queue Example 2.32.0
[INFO] from pom.xml
[INFO] -----------------------------------[ jar ]-----------------------------------
[INFO] [INFO] --- enforcer:3.3.0:enforce (enforce-maven-version) @ queue ---
[INFO] [INFO] --- enforcer:3.3.0:enforce (enforce-java-version) @ queue ---
[INFO] [INFO] --- remote-resources:3.1.0:process (process-resource-bundles) @ queue ---
[INFO] Copying 3 resources from 1 bundle.
[INFO] [INFO] --- resources:3.3.1:resources (default-resources) @ queue ---
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO] Copying 3 resources from target/maven-shared-archive-resources to target/classes
[INFO] [INFO] --- compiler:3.11.0:compile (default-compile) @ queue ---
[INFO] Changes detected - recompiling the module! :source
[INFO] Compiling 1 source file with javac [debug target 11] to target/classes
[INFO] [INFO] --- resources:3.3.1:testResources (default-testResources) @ queue ---
[INFO] skip non existing resourceDirectory /home/user/activemq-artemis-
```
INFO Copying 3 resources from target/maven-shared-archive-resources to target/test-classes

INFO --- compiler:3.11.0:testCompile (default-testCompile) @ queue ---
INFO No sources to compile

INFO --- surefire:2.22.2:test (default-test) @ queue ---

INFO --- jar:3.3.0:jar (default-jar) @ queue ---
INFO Building jar: /home/user/activemq-artemis-examples/examples/features/standard/queue/target/queue-2.32.0.jar

INFO --- site:3.12.1:attach-descriptor (attach-descriptor) @ queue ---
INFO Skipping because packaging 'jar' is not pom.

INFO --- source:3.3.0:jar-no-fork (attach-sources) @ queue ---
INFO Building jar: /home/user/activemq-artemis-examples/examples/features/standard/queue/target/queue-2.32.0-sources.jar

INFO --- artemis:2.32.0:create (create) @ queue ---
INFO --- artemis:2.32.0:cli (start) @ queue ---
INFO awaiting server to start

INFO: Apache ActiveMQ Artemis 2.32.0

INFO awaiting server to start
INFO: 2023-12-14 14:33:06,314 INFO [org.apache.activemq.artemis.integration.bootstrap] AMQ101000: Starting ActiveMQ Artemis Server version 2.32.0
INFO: 2023-12-14 14:33:06,380 INFO [org.apache.activemq.artemis.core.server] AMQ221057: Global Max Size is being adjusted to 1/2 of the JVM max size (-Xmx). being defined as 1073741824
AMQ221043: Protocol module found: [artemis-hornetq-protocol]. Adding protocol support for: HORNETQ
server-out:2023-12-14 14:33:06,411 INFO [org.apache.activemq.artemis.core.server] AMQ221034: Waiting indefinitely to obtain primary lock
server-out:2023-12-14 14:33:06,411 INFO [org.apache.activemq.artemis.core.server] AMQ221035: Primary Server Obtained primary lock
server-out:2023-12-14 14:33:06,450 INFO [org.apache.activemq.artemis.core.server] AMQ221080: Deploying address DLQ supporting [ANYCAST]
server-out:2023-12-14 14:33:06,454 INFO [org.apache.activemq.artemis.core.server] AMQ221003: Deploying ANYCAST queue DLQ on address DLQ
server-out:2023-12-14 14:33:06,476 INFO [org.apache.activemq.artemis.core.server] AMQ221003: Deploying ANYCAST queue ExpiryQueue on address ExpiryQueue
server-out:2023-12-14 14:33:06,681 INFO [org.apache.activemq.artemis.core.server] AMQ221020: Started EPOLL Acceptor at 0.0.0.0:61616 for protocols [CORE,MQTT,AMQP,STOMP,HORNETQ,OPENWIRE]
server-out:2023-12-14 14:33:06,683 INFO [org.apache.activemq.artemis.core.server] AMQ221020: Started EPOLL Acceptor at 0.0.0.0:5445 for protocols [HORNETQ,STOMP]
server-out:2023-12-14 14:33:06,685 INFO [org.apache.activemq.artemis.core.server] AMQ221020: Started EPOLL Acceptor at 0.0.0.0:5672 for protocols [AMQP]
server-out:2023-12-14 14:33:06,687 INFO [org.apache.activemq.artemis.core.server] AMQ221020: Started EPOLL Acceptor at 0.0.0.0:1883 for protocols [MQTT]
server-out:2023-12-14 14:33:06,689 INFO [org.apache.activemq.artemis.core.server] AMQ221020: Started EPOLL Acceptor at 0.0.0.0:61613 for protocols [STOMP]
server-out:2023-12-14 14:33:06,690 INFO [org.apache.activemq.artemis.core.server] AMQ221007: Server is now live
server-out:2023-12-14 14:33:06,690 INFO [org.apache.activemq.artemis.core.server] AMQ221001: Apache ActiveMQ Artemis Message Broker version 2.32.0 [0.0.0.0, nodeID =b20d82dd-9a8d-11ee-a56d-000c2997e711]
[INFO] Server started
[INFO]
[INFO] --- artemis:2.32.0:runClient (runClient) @ queue ---
Sent message: This is a text message
Received message: This is a text message
[INFO]
[INFO] --- artemis:2.32.0:stop (stop) @ queue ---
server-out:2023-12-14 14:33:08,295 INFO [org.apache.activemq.artemis.core.server] AMQ221002: Apache ActiveMQ Artemis Message Broker version 2.32.0 [b20d82dd-9a8d-11ee-a56d-000c2997e711] stopped, uptime 1.969 seconds
server-out:Server stopped!
[INFO] ------------------------------------------------------------------------
This list includes a preview of some examples that exist for Artemis in the Examples Repository, see the repository for more.

87.2. Application-Layer Failover

Apache ActiveMQ Artemis also supports Application-Layer failover, useful in the case that replication is not enabled on the server side.

With Application-Layer failover, it’s up to the application to register a JMS ExceptionListener with Apache ActiveMQ Artemis which will be called by Apache ActiveMQ Artemis in the event that connection failure is detected.

The code in the ExceptionListener then recreates the JMS connection, session, etc on another node and the application can continue.

Application-layer failover is an alternative approach to High Availability (HA). Application-layer failover differs from automatic failover in that some client side coding is required in order to implement this. Also, with Application-layer failover, since the old session object dies and a new one is created, any uncommitted work in the old session will be lost, and any unacknowledged messages might be redelivered.

87.3. Core Bridge Example

The bridge example demonstrates a core bridge deployed on one server, which consumes messages from a local queue and forwards them to an address on a second server.

Core bridges are used to create message flows between any two Apache ActiveMQ Artemis servers which are remotely separated. Core bridges are resilient and will cope with temporary connection failure allowing them to be an ideal choice for forwarding over unreliable connections, e.g. a WAN.

87.4. Browser

The browser example shows you how to use a JMS QueueBrowser with Apache ActiveMQ Artemis.

Queues are a standard part of JMS, please consult the JMS 2.0 specification for full details.

A QueueBrowser is used to look at messages on the queue without removing them. It can scan the entire content of a queue or only messages matching a message selector.
87.5. Camel

The camel example demonstrates how to build and deploy a Camel route to the broker using a web application archive (i.e. war file).

87.6. Client Kickoff

The client-kickoff example shows how to terminate client connections given an IP address using the JMX management API.

87.7. Client side failover listener

The client-side-failoverlistener example shows how to register a listener to monitor failover events

87.8. Client-Side Load-Balancing

The client-side-load-balancing example demonstrates how sessions created from a single JMS Connection can be created to different nodes of the cluster. In other words it demonstrates how Apache ActiveMQ Artemis does client-side load-balancing of sessions across the cluster.

87.9. Clustered Durable Subscription

This example demonstrates a clustered JMS durable subscription

87.10. Clustered Grouping

This is similar to the message grouping example except that it demonstrates it working over a cluster. Messages sent to different nodes with the same group id will be sent to the same node and the same consumer.

87.11. Clustered Queue

The clustered-queue example demonstrates a queue deployed on two different nodes. The two nodes are configured to form a cluster. We then create a consumer for the queue on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both consumers receive the sent messages in a round-robin fashion.

87.12. Clustering with JGroups

The clustered-jgroups example demonstrates how to form a two node cluster using JGroups as its underlying topology discovery technique, rather than the default UDP broadcasting. We then create a consumer for the queue on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both consumers receive the sent messages in a round-robin fashion.
87.13. Clustered Standalone

The clustered-standalone example demonstrates how to configure and starts 3 cluster nodes on the same machine to form a cluster. A subscriber for a JMS topic is created on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that the 3 subscribers receive all the sent messages.

87.14. Clustered Static Discovery

This example demonstrates how to configure a cluster using a list of connectors rather than UDP for discovery.

87.15. Clustered Static Cluster One Way

This example demonstrates how to set up a cluster where cluster connections are one way, i.e. server A -> Server B -> Server C.

87.16. Clustered Topic

The clustered-topic example demonstrates a JMS topic deployed on two different nodes. The two nodes are configured to form a cluster. We then create a subscriber on the topic on each node, and we create a producer on only one of the nodes. We then send some messages via the producer, and we verify that both subscribers receive all the sent messages.

87.17. Message Consumer Rate Limiting

With Apache ActiveMQ Artemis you can specify a maximum consume rate at which a JMS MessageConsumer will consume messages. This can be specified when creating or deploying the connection factory.

If this value is specified then Apache ActiveMQ Artemis will ensure that messages are never consumed at a rate higher than the specified rate. This is a form of consumer throttling.

87.18. Dead Letter

The dead-letter example shows you how to define and deal with dead letter messages. Messages can be delivered unsuccessfully (e.g. if the transacted session used to consume them is rolled back).

Such a message goes back to the JMS destination ready to be redelivered. However, this means it is possible for a message to be delivered again and again without any success and remain in the destination, clogging the system.

To prevent this, messaging systems define dead letter messages: after a specified unsuccessful delivery attempts, the message is removed from the destination and put instead in a dead letter destination where they can be consumed for further investigation.
87.19. Delayed Redelivery

The delayed-redelivery example demonstrates how Apache ActiveMQ Artemis can be configured to provide a delayed redelivery in the case a message needs to be redelivered.

Delaying redelivery can often be useful in the case that clients regularly fail or roll-back. Without a delayed redelivery, the system can get into a “thrashing” state, with delivery being attempted, the client rolling back, and delivery being re-attempted in quick succession, using up valuable CPU and network resources.

87.20. Divert

Apache ActiveMQ Artemis diverts allow messages to be transparently "diverted" or copied from one address to another with just some simple configuration defined on the server side.

87.21. Durable Subscription

The durable-subscription example shows you how to use a durable subscription with Apache ActiveMQ Artemis. Durable subscriptions are a standard part of JMS, please consult the JMS 1.1 specification for full details.

Unlike non-durable subscriptions, the key function of durable subscriptions is that the messages contained in them persist longer than the lifetime of the subscriber - i.e. they will accumulate messages sent to the topic even if there is no active subscriber on them. They will also survive server restarts or crashes. Note that for the messages to be persisted, the messages sent to them must be marked as durable messages.

87.22. Embedded

The embedded example shows how to embed a broker within your own code using POJO instantiation and no config files.

87.23. Embedded Simple

The embedded-simple example shows how to embed a broker within your own code using regular Apache ActiveMQ Artemis XML files.

87.24. Exclusive Queue

The exclusive-queue example shows you how to use exclusive queues, that route all messages to only one consumer at a time.

87.25. Message Expiration

The expiry example shows you how to define and deal with message expiration. Messages can be retained in the messaging system for a limited period of time before being removed. JMS
specification states that clients should not receive messages that have been expired (but it does not guarantee this will not happen).

Apache ActiveMQ Artemis can assign an expiry address to a given queue so that when messages are expired, they are removed from the queue and sent to the expiry address. These "expired" messages can later be consumed from the expiry address for further inspection.

87.26. Apache ActiveMQ Artemis Resource Adapter example

This examples shows how to build the activemq resource adapters a rar for deployment in other Application Server's

87.27. HTTP Transport

The http-transport example shows you how to configure Apache ActiveMQ Artemis to use the HTTP protocol as its transport layer.

87.28. Instantiate JMS Objects Directly

Usually, JMS Objects such as ConnectionFactory, Queue and Topic instances are looked up from JNDI before being used by the client code. This objects are called "administered objects" in JMS terminology.

However, in some cases a JNDI server may not be available or desired. To come to the rescue Apache ActiveMQ Artemis also supports the direct instantiation of these administered objects on the client side so you don't have to use JNDI for JMS.

87.29. Interceptor

Apache ActiveMQ Artemis allows an application to use an interceptor to hook into the messaging system. Interceptors allow you to handle various message events in Apache ActiveMQ Artemis.

87.30. Interceptor AMQP

Similar to the Interceptor example, but using AMQP interceptors.

87.31. Interceptor Client

Similar to the Interceptor example, but using interceptors on the client rather than the broker.

87.32. Interceptor MQTT

Similar to the Interceptor example, but using MQTT interceptors.
87.33. **JAAS**

The `jaas` example shows you how to configure Apache ActiveMQ Artemis to use JAAS for security. Apache ActiveMQ Artemis can leverage JAAS to delegate user authentication and authorization to existing security infrastructure.

87.34. **JMS Auto Closable**

The `jms-auto-closeable` example shows how JMS resources, such as connections, sessions and consumers, in JMS 2 can be automatically closed on error.

87.35. **JMS Completion Listener**

The `jms-completion-listener` example shows how to send a message asynchronously to Apache ActiveMQ Artemis and use a CompletionListener to be notified of the Broker receiving it.

87.36. **JMS Bridge**

The `jms-bridge` example shows how to setup a bridge between two standalone Apache ActiveMQ Artemis servers.

87.37. **JMS Context**

The `jms-context` example shows how to send and receive a message to/from an address/queue using Apache ActiveMQ Artemis by using a JMS Context.

A JMSCContext is part of JMS 2.0 and combines the JMS Connection and Session Objects into a simple Interface.

87.38. **JMS Shared Consumer**

The `jms-shared-consumer` example shows you how can use shared consumers to share a subscription on a topic. In JMS 1.1 this was not allowed and so caused a scalability issue. In JMS 2 this restriction has been lifted so you can share the load across different threads and connections.

87.39. **JMX Management**

The `jmx` example shows how to manage Apache ActiveMQ Artemis using JMX.

87.40. **Large Message**

The `large-message` example shows you how to send and receive very large messages with Apache ActiveMQ Artemis. Apache ActiveMQ Artemis supports the sending and receiving of huge messages, much larger than can fit in available RAM on the client or server. Effectively the only limit to message size is the amount of disk space you have on the server.
Large messages are persisted on the server so they can survive a server restart. In other words Apache ActiveMQ Artemis doesn’t just do a simple socket stream from the sender to the consumer.

87.41. Last-Value Queue

The last-value-queue example shows you how to define and deal with last-value queues. Last-value queues are special queues which discard any messages when a newer message with the same value for a well-defined last-value property is put in the queue. In other words, a last-value queue only retains the last value.

A typical example for last-value queue is for stock prices, where you are only interested by the latest price for a particular stock.

87.42. Management

The management example shows how to manage Apache ActiveMQ Artemis using JMS Messages to invoke management operations on the server.

87.43. Management Notification

The management-notification example shows how to receive management notifications from Apache ActiveMQ Artemis using JMS messages. Apache ActiveMQ Artemis servers emit management notifications when events of interest occur (consumers are created or closed, addresses are created or deleted, security authentication fails, etc.).

87.44. Message Counter

The message-counters example shows you how to use message counters to obtain message information for a queue.

87.45. Message Group

The message-group example shows you how to configure and use message groups with Apache ActiveMQ Artemis. Message groups allow you to pin messages so they are only consumed by a single consumer. Message groups are sets of messages that has the following characteristics:

- Messages in a message group share the same group id, i.e. they have same JMSXGroupID string property values
- The consumer that receives the first message of a group will receive all the messages that belongs to the group

87.46. Message Group

The message-group2 example shows you how to configure and use message groups with Apache ActiveMQ Artemis via a connection factory.
87.47. Message Priority

Message Priority can be used to influence the delivery order for messages. It can be retrieved by the message's standard header field 'JMSPriority' as defined in JMS specification version 1.1. The value is of type integer, ranging from 0 (the lowest) to 9 (the highest). When messages are being delivered, their priorities will effect their order of delivery. Messages of higher priorities will likely be delivered before those of lower priorities.

Messages of equal priorities are delivered in the natural order of their arrival at their destinations. Please consult the JMS 1.1 specification for full details.

87.48. Multiple Failover

This example demonstrates how to set up a primary server with multiple backups

87.49. Multiple Failover Failback

This example demonstrates how to set up a primary server with multiple backups but forcing failover back to the original primary server

87.50. No Consumer Buffering

By default, Apache ActiveMQ Artemis consumers buffer messages from the server in a client side buffer before you actually receive them on the client side. This improves performance since otherwise every time you called receive() or had processed the last message in a MessageListener onMessage() method, the Apache ActiveMQ Artemis client would have to go to the server to request the next message, which would then get sent to the client side, if one was available.

This would involve a network round trip for every message and reduce performance. Therefore, by default, Apache ActiveMQ Artemis pre-fetches messages into a buffer on each consumer.

In some case buffering is not desirable, and Apache ActiveMQ Artemis allows it to be switched off. This example demonstrates that.

87.51. Non-Transaction Failover With Server Data Replication

The non-transaction-failover example demonstrates two servers coupled as a live-backup pair for high availability (HA), and a client using a non-transacted JMS session failing over from primary to backup when the primary server is crashed.

Apache ActiveMQ Artemis implements failover of client connections between primary and backup servers. This is implemented by the replication of state between primary and backup nodes. When replication is configured and a primary node crashes, the client connections can carry on and
continue to send and consume messages. When non-transacted sessions are used, once and only once message delivery is not guaranteed and it is possible that some messages will be lost or delivered twice.

87.52. OpenWire

The Openwire example shows how to configure an Apache ActiveMQ Artemis server to communicate with an ApacheActiveMQ Artemis JMS client that uses open-wire protocol.

You will find the queue example for open wire, and the chat example. The virtual-topic-mapping examples shows how to map the ActiveMQ "Classic" Virtual Topic naming convention to work with the Artemis Address model.

87.53. Paging

The paging example shows how Apache ActiveMQ Artemis can support huge queues even when the server is running in limited RAM. It does this by transparently paging messages to disk, and depaging them when they are required.

87.54. Pre-Acknowledge

Standard JMS supports three acknowledgement modes: AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, and DUPS_OK_ACKNOWLEDGE. For a full description on these modes please consult the JMS specification, or any JMS tutorial.

All of these standard modes involve sending acknowledgements from the client to the server. However in some cases, you really don’t mind losing messages in event of failure, so it would make sense to acknowledge the message on the server before delivering it to the client. This example demonstrates how Apache ActiveMQ Artemis allows this with an extra acknowledgement mode.

87.55. Message Producer Rate Limiting

The producer-rte-limit example demonstrates how, with Apache ActiveMQ Artemis, you can specify a maximum send rate at which a JMS message producer will send messages.

87.56. Queue

A simple example demonstrating a queue.

87.57. Message Redistribution

The queue-message-redistribution example demonstrates message redistribution between queues with the same name deployed in different nodes of a cluster.
87.58. Queue Requestor

A simple example demonstrating a JMS queue requestor.

87.59. Queue with Message Selector

The queue-selector example shows you how to selectively consume messages using message selectors with queue consumers.

87.60. Reattach Node example

The Reattach Node example shows how a client can try to reconnect to the same server instead of failing the connection immediately and notifying any user ExceptionListener objects. Apache ActiveMQ Artemis can be configured to automatically retry the connection, and reattach to the server when it becomes available again across the network.

87.61. Replicated Failback example

An example showing how failback works when using replication. In this example a primary server will replicate all its Journal to a backup server as it updates it. When the primary server crashes the backup takes over from the primary server and the client reconnects and carries on from where it left off.

87.62. Replicated Failback static example

An example showing how failback works when using replication, but this time with static connectors

87.63. Replicated multiple failover example

An example showing how to configure multiple backups when using replication

87.64. Replicated Failover transaction example

An example showing how failover works with a transaction when using replication

87.65. Request-Reply example

A simple example showing the JMS request-response pattern.

87.66. Scheduled Message

The scheduled-message example shows you how to send a scheduled message to an address/queue with Apache ActiveMQ Artemis. Scheduled messages won't get delivered until a specified time in the future.
87.67. Security

The security example shows you how configure and use role based security with Apache ActiveMQ Artemis.

87.68. Security LDAP

The security-ldap example shows you how configure and use role based security with Apache ActiveMQ Artemis & an embedded instance of the Apache DS LDAP server.

87.69. Security keycloak

The security-keycloak example shows you how to delegate security with Apache ActiveMQ Artemis & an external Keycloak. Using OAuth of the web console and direct access for JMS clients.

87.70. Send Acknowledgements

The send-acknowledgements example shows you how to use Apache ActiveMQ Artemis's advanced asynchronous send acknowledgements feature to obtain acknowledgement from the server that sends have been received and processed in a separate stream to the sent messages.

87.71. Slow Consumer

The slow-consumer example shows you how to detect slow consumers and configure a slow consumer policy in Apache ActiveMQ Artemis's

87.72. Spring Integration

This example shows how to use embedded JMS using Apache ActiveMQ Artemis's Spring integration.

87.73. SSL Transport

The ssl-enabled shows you how to configure SSL with Apache ActiveMQ Artemis to send and receive message.

87.74. Static Message Selector

The static-selector example shows you how to configure an Apache ActiveMQ Artemis core queue with static message selectors (filters).

87.75. Static Message Selector Using JMS

The static-selector-jms example shows you how to configure an Apache ActiveMQ Artemis queue with static message selectors (filters) using JMS.
87.76. Stomp

The *stomp* example shows you how to configure an Apache ActiveMQ Artemis server to send and receive Stomp messages.

87.77. Stomp1.1

The *stomp* example shows you how to configure an Apache ActiveMQ Artemis server to send and receive Stomp messages via a Stomp 1.1 connection.

87.78. Stomp1.2

The *stomp* example shows you how to configure an Apache ActiveMQ Artemis server to send and receive Stomp messages via a Stomp 1.2 connection.

87.79. Stomp Over Web Sockets

The *stomp-websockets* example shows you how to configure an Apache ActiveMQ Artemis server to send and receive Stomp messages directly from Web browsers (provided they support Web Sockets).

87.80. Symmetric Cluster

The *symmetric-cluster* example demonstrates a symmetric cluster set-up with Apache ActiveMQ Artemis.

Apache ActiveMQ Artemis has extremely flexible clustering which allows you to set-up servers in many different topologies. The most common topology that you’ll perhaps be familiar with if you are used to application server clustering is a symmetric cluster.

With a symmetric cluster, the cluster is homogeneous, i.e. each node is configured the same as every other node, and every node is connected to every other node in the cluster.

87.81. Temporary Queue

A simple example demonstrating how to use a JMS temporary queue.

87.82. Topic

A simple example demonstrating a JMS topic.

87.83. Topic Hierarchy

Apache ActiveMQ Artemis supports topic hierarchies. With a topic hierarchy you can register a subscriber with a wild-card and that subscriber will receive any messages sent to an address that matches the wild card.
87.84. Topic Selector 1

The `topic-selector-example1` example shows you how to send message to a JMS Topic, and subscribe them using selectors with Apache ActiveMQ Artemis.

87.85. Topic Selector 2

The `topic-selector-example2` example shows you how to selectively consume messages using message selectors with topic consumers.

87.86. Transaction Failover

The `transaction-failover` example demonstrates two servers coupled as a live-backup pair for high availability (HA), and a client using a transacted JMS session failing over from primary to backup when the primary server is crashed.

Apache ActiveMQ Artemis implements failover of client connections between primary and backup servers. This is implemented by the sharing of a journal between the servers. When a primary node crashes, the client connections can carry and continue to send and consume messages. When transacted sessions are used, once and only once message delivery is guaranteed.

87.87. Failover Without Transactions

The `stop-server-failover` example demonstrates failover of the JMS connection from one node to another when the primary server crashes using a JMS non-transacted session.

87.88. Transactional Session

The `transactional` example shows you how to use a transactional Session with Apache ActiveMQ Artemis.

87.89. XA Heuristic

The `xa-heuristic` example shows you how to make an XA heuristic decision through Apache ActiveMQ Artemis Management Interface. A heuristic decision is a unilateral decision to commit or rollback an XA transaction branch after it has been prepared.

87.90. XA Receive

The `xa-receive` example shows you how message receiving behaves in an XA transaction in Apache ActiveMQ Artemis.

87.91. XA Send

The `xa-send` example shows you how message sending behaves in an XA transaction in Apache
ActiveMQ Artemis.
Chapter 88. Legal Notice

Licensed to the Apache Software Foundation (ASF) under one or more contributor license agreements. See the NOTICE file distributed with this work for additional information regarding copyright ownership. The ASF licenses this file to You under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.