Fuse Message Broker

Fault Tolerant Messaging

Version 5.5
Febuary 2012

Fault Tolerant Messaging

Version 5.5

Updated: 27 Mar 2014

Trademark Disclaimer

Third Party Acknowledgements

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (http://jline.sourceforge.net) jline:jline:jar:1.0

  License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux `<mwp1@cornell.edu>`

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (http://woodstox.codehaus.org/StAX2) org.codehaus.woodstox:stax2-api:jar:3.1.1

  License: The BSD License (http://www.opensource.org/licenses/bsd-license.php)

  Copyright (c) <YEAR>, <OWNER> All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (http://www.jibx.org/main-reactor/jibx-run) org.jibx:jibx-run:bundle:1.2.3

  License: BSD (http://jibx.sourceforge.net/jibx-license.html) Copyright (c) 2003-2010, Dennis M. Sosnoski.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (http://www.jboss.org/javassist) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile

  License: MPL (http://www.mozilla.org/MPL/MPL-1.1.html)

- HAPI-OSGI-Base Module (http://hl7api.sourceforge.net/hapi-osgi-base/) ca.uhn.hapi:hapi-osgi-base:bundle:1.2

  License: Mozilla Public License 1.1 (http://www.mozilla.org/MPL/MPL-1.1.txt)

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction

Fault tolerant message systems can recover from failures with little or no interruption of functionality. Fuse Message Broker does this by making it easy to configure clients to fail over to new brokers in the event of a broker failure. It also makes it easy to set up master/slave groups that allow brokers to take over for each other and maintain the integrity of persistent messages and transactions.

Overview

If planned for, disaster scenarios that result in the loss of a message broker need not obstruct message delivery. Making a messaging system fault tolerant involves:

- deploying multiple brokers into a topology that allows one broker to pick up the duties of a failed broker
- configuring clients to fail over to a new broker in the event that its current broker fails

Fuse Message Broker provides mechanisms that make building fault tolerant messaging systems easy.

Failover protocol

The failover protocol allows you to configure a client with a list of brokers to which it can connect. When one broker fails, a clients using the failover protocol will automatically reconnect to a new broker from its list. As long as one of the brokers on the list is running, the client can continue to function uninterrupted.

When combined with brokers deployed in a master/slave topology, the failover protocol is a key part of a fault-tolerant messaging system. The clients will automatically fail over to the slave broker if the master fails. The clients will remain functional and continue working as if nothing had happened.

For more information, see "Failover Protocol" on page 14.

Master/Slave topologies

A *master/slave* topology includes a master broker and one or more slave brokers. All of the brokers share data by using either a store and forward mechanism or by using a shared data store. When the master broker fails, one of the slave brokers takes over and becomes the new master broker. Client applications can reconnect to the new master broker and resume processing as normal.

For details, see "Master/Slave" on page 31.

Chapter 2. Client Failover

Fuse Message Broker provides two simple mechanisms for clients to failover to an alternate broker if the current connection fails. The failover protocol relies on either a hard coded list of brokers or a broker participating in a network of brokers to provide the list of alternate brokers. The discovery protocol relies on discovery agents to provide a list of alternative brokers.

Failover Protocol

The *failover protocol* facilitates quick recovery from network failures. When a recoverable network error occurs the protocol catches the error and automatically attempts to reestablish the JMS connection to an alternate broker endpoint without the need to recreate all of the JMS objects associated with the JMS connection. The failover URI is composed of one or more URIs that represent different broker endpoints. By default, the protocol randomly chooses a URI from the list and attempts to establish a network connection to it. If it does not succeed, or if it subsequently fails, a new network connection is established to one of the other URIs in the list.

For true high-availability and fail over capabilities, you will need to set up your brokers in a network of brokers. See *Using Networks of Brokers*.

You can set up failover in one of the following ways:

- Static—the client is configured with a static list of available URIs
- Dynamic—the brokers push information about the available broker connections

Static Failover

Overview

In static failover a client is configured to use a *failover IRU* that lists the URIs of the broker connections the client can use. When establishing a connection, the client randomly chooses a URI from the list and attempts to establish a connection to it. If the connection does not succeed, the client chooses a new URI from the list and tries again. The client will continue cycling through the list until a connection attempt succeeds.

If a client's connection to a broker fails after it has been established, the client will attempt to reconnect to a different broker in the list. Once a connection to a new broker is established, the client will continue to use the new broker until the connection to the new broker is severed.

Failover URI

A failover URI is a composite URI that uses one of the following syntaxes:

- `failover://uri1,...,uriN`
- `failover://(uri1,...,uriN)?TransportOptions`

The URI list(`uri1,...,uriN`) is a comma-separated list containing the list of broker endpoint URIs to which the client can connect. The transport options(`?TransportOptions`) specified in the form of a query list, allow you to configure some of the failoiver behaviors.

Transport options

The failover protocol supports the transport options described in Table 2.1 on page 15.

Table 2.1. Failover Transport Options

Option	Default	Description
`initialReconnectDelay`	10	Specifies the number of milliseconds to wait before the first reconnect attempt.
`maxReconnectDelay`	30000	Specifies the maximum amount of time, in milliseconds, to wait between reconnect attempts.
`useExponentialBackOff`	true	Specifies whether to use an exponential back-off between reconnect attempts.

Option	Default	Description
`backOffMultiplier`	2	Specifies the exponent used in the exponential back-off algorithm.
`maxReconnectAttempts`	-1	Specifies the maximum number of reconnect attempts before an error is returned to the client. -1 specifies unlimited attempts. 0 specifies that an initial connection attempt is made at start-up, but no attempts to failover over to a secondary broker will be made.
`startupMaxReconnectAttempts`	0	Specifies the maximum number of reconnect attempts before an error is returned to the client on the *first* attempt by the client to start a connection. 0 specifies unlimited attempts.
`randomize`	true	Specifies if a URI is chosen at random from the list. Otherwise, the list is traversed from left to right.
`backup`	false	Specifies if the protocol initializes and holds a second transport connection to enable fast failover.
`timeout`	-1	Specifies the amount of time, in milliseconds, to wait before sending an error if a new connection is not established. -1 specifies an infinite timeout value.
`trackMessages`	false	Specifies if the protocol keeps a cache of in-flight messages that are flushed to a broker on reconnect.
`maxCacheSize`	131072	Specifies the size, in bytes, used for the cache used to track messages.
`updateURIsSupported`	true	Specifies whether the client accepts updates to its list of known URIs from the connected broker. Setting this to false inhibits the client's ability to use dynamic failover. See "Dynamic Failover" on page 18.

Example

Example 2.1 on page 17 shows a failover URI that can connect to one of two message brokers.

Example 2.1. Simple Failover URI

```
failover://(tcp://localhost:61616,tcp://remotehost:61616)?initialReconnectDelay=100
```

Dynamic Failover

Overview

Dynamic failover combines the failover protocol and a network of brokers to allow a broker to supply its clients with a list of broker connections to which the clients can failover. Clients use a failover URI to connect to a broker and the broker dynamically updates the clients' list of available URIs. The broker updates its clients' failover lists with the URIs of the other brokers in its network of brokers that are currently running. As new brokers join, or exit, the cluster, the broker will adjust its clients' failover lists.

From a connectivity point of view, dynamic failover works the same as static failover. A client randomly chooses a URI from the list provided in its failover URI. Once that connection is established, the list of available brokers is updated. If the original connection fails, the client will randomly select a new URI from its dynamically generated list of brokers. If the new broker is configured for dynamic failover, the new broker will update the client's list.

Set-up

To use dynamic failover you must configure both the clients and brokers used by your application. The following must be configured:

- The client's must be configured to use the failover protocol when connecting with its broker.
- The brokers must be configured to form a network of brokers.

  See *Using Networks of Brokers*.
- The broker's transport connector must set the failover properties needed to update its consumers.

Client-side configuration

The client-side configuration for using dynamic failover is identical to the client-side configuration for using static failover. The client uses a failover URI to connect to a broker.

When using dynamic failover, the failover URI can include a single broker URI. As long as the broker is available when the client attempts to make its initial connection, the client's list of failover brokers will get populated.

It is also important that the `updateURIsSupported` option not be set to `false`. If it is, the client will not be able to receive updates about what brokers are available for failover.

See "Failover URI" on page 15 and "Transport options" on page 15 for more information about using failover URIs.

Broker-side configuration

Important

Brokers should *never* use a failover uri to configure a transport connector. The failover protocol does not support listening for incoming messages.

Configuring a broker to participate in dynamic failover requires two things:

- The broker must be configured to participate in a network of brokers that can be available for failovers.

  See *Using Networks of Brokers* for information about setting up a network of brokers.

- The broker's transport connector must set the failover properties needed to update its consumers.

Table 2.2 on page 19 describes the broker-side properties that can be used to configure a failover cluster. These properties are attributes on the broker's `transportConnector` element.

Table 2.2. Broker-side Failover Properties

Property	Default	Description
`updateClusterClients`	`false`	Specifies if the broker passes information to connected clients about changes in the topology of the broker cluster.
`updateClusterClientsOnRemove`	`false`	Specifies if clients are updated when a broker is removed from the cluster.
`rebalanceClusterClients`	`false`	Specifies if connected clients are asked to rebalance across the cluster whenever a new broker joins.
`updateClusterFilter`		Specifies a comma-separated list of regular expression filters, which match against broker names to select the brokers that belong to the failover cluster.
`updateURIsURL`		Specifies a URL, or path to a local file, locating a text file that contains a comma-separated list of URIs to

Property	Default	Description
		use for reconnect in the case of failure.

Example

Example 2.2 on page 20 shows the configuration for a broker that participates in dynamic failover.

Example 2.2. Broker for Dynamic Failover

```
<beans ... >
  <broker>
    ...
    <networkConnectors>
❶       <networkConnector uri="multicast://default" />
    </networkConnectors>
    ...
    <transportConnectors>
      <transportConnector name="openwire"
          uri="tcp://0.0.0.0:61616"
❷         discoveryUri="multicast://default"
❸         updateClusterClients="true"
❹         updateClusterFilter="*A*,*B*" />
    </transportConnectors>
    ...
  </broker>
</beans>
```

The configuration in Example 2.2 on page 20 does the following:

❶ Creates a network connector that connects to any discoverable broker that uses the multicast transport.
❷ Makes the broker discoverable by other brokers over the multicast protocol.
❸ Makes the broker update the list of available brokers for clients that connect using the failover protocol.

> **Note**
> Clients will only be updated when new brokers join the cluster, not when a broker leaves the cluster.

❹ Creates a filter so that only those brokers whose names start with the letter `A` or the letter `B` are considered to belong to the failover cluster.

Example 2.3 on page 21 shows the URI for a client that uses the failover protocol to connect to the broker and its cluster.

Example 2.3. Failover URI for Connecting to a Failover Cluster

```
failover://(tcp://0.0.0.0:61616)?initialReconnectDelay=100
```

Discovery Protocol

Dynamic failover provides a lot of control over how a client generates its list of available brokers, but it has weaknesses. It requires that you know the address of the initial broker and that the initial broker is active when the client starts up. It also requires that all of the brokers being used for failover are configured in a network of brokers.

Fuse Message Broker's discovery protocol offers an alternative method for dynamically generating a list of brokers that are available for client failover. The protocol feature allows brokers to advertise their availability and for clients to dynamically discover them. This is accomplished using two pieces:

- *discovery agents*—components that advertise the list of available brokers
- *discovery URI*—a URI that looks up all of the discoverable brokers and presents them as a list of actual URIs for use by the client or network connector

Discovery Agents

A discovery agent is a mechanism that advertises available brokers to clients and other brokers. When a client, or broker, using a discovery URI starts up it will look for any brokers that are available using the specified discovery agent. The clients will update their lists periodically using the same mechanism.

How a discover agent learns about the available brokers varies between agents. Some agents use a static list, some use a third party registry, and some rely on the brokers to provide the information. For discovery agents that rely on the brokers for information, it is necessary to enable the discovery agent in the message broker configuration. For example, to enable the multicast discovery agent on an Openwire endpoint, you edit the relevant `transportConnector` element as shown in Example 2.4 on page 23.

Example 2.4. Enabling a Discovery Agent on a Broker

```
<transportConnectors>
  <transportConnector name="openwire"
    uri="tcp://localhost:61716"
    discoveryUri="multicast://default" />
</transportConnectors>
```

Where the `discoveryUri` attribute on the `transportConnector` element is initialized to `multicast://default`.

Tip

If a broker uses multiple transport connectors, you need to configure each transport connector to use a discovery agent individually. This means that different connectors can use different discovery mechanisms or that one or more of the connectors can be indiscoverable.

Fuse Message Broker currently supports the following discovery agents:

- Static Discovery Agent
- Multicast Discovery Agent
- Zeroconf Discovery Agent

Static Discovery Agent

Overview

The *static discovery agent* does not truly discover the available brokers. It uses an explicit list of broker URLs to specify the available brokers. Brokers are not involved with the static discovery agent. The client only knows about the brokers that are hard coded into the agent's URI.

Using the agent

The static discovery agent is a client-side only agent. It does not require any configuration on the brokers that will be discovered.

To use the agent, you simply configure the client to connect to a broker using a discovery protocol that uses a static agent URI.

The static discovery agent URI conforms to the syntax in Example 2.5 on page 24.

Example 2.5. Static Discovery Agent URI Format

```
static://(URI1,URI2,URI3,...)
```

Example

Example 2.6 on page 24 shows a URL that configures a client to use the dynamic discovery protocol to connect to one member of a broker pair.

Example 2.6. Discovery URL using the Static Discovery Agent

```
discovery://(static://(tcp://localhost:61716,tcp://localhost:61816))
```

Multicast Discovery Agent

Overview

The *multicast discovery agent* uses the IP multicast protocol to find any message brokers currently active on the local network. The agent requires that *each* broker you want to advertise is configured to use the multicast agent to publish its details to a multicast group. Clients using the multicast agent as part of the discovery URI they use for connecting to a broker will use the agent to receive the list of available brokers advertising in the specified multicast group.

Important

Your local network (LAN) must be configured appropriately for the IP/multicast protocol to work.

URI

The multicast discovery agent URI conforms to the syntax in Example 2.7 on page 25.

Example 2.7. Multicast Discovery Agent URI Format

```
multicast://GroupID
```

Where *`GroupID`* is an alphanumeric identifier. All participants in the same discovery network must use the same *`GroupID`*.

Configuring a broker

For a broker to be discoverable using the multicast discovery agent, you must enable the discovery agent in the broker's configuration. To enable the multicast discovery agent you set the `transportConnector` element's `discoveryUri` attribute to a mulitcast discovery agent URI as shown in Example 2.8 on page 25.

Example 2.8. Enabling a Multicast Discovery Agent on a Broker

```
<transportConnectors>
  <transportConnector name="openwire"
    uri="tcp://localhost:61716"
    discoveryUri="multicast://default" />
</transportConnectors>
```

The broker configured in Example 2.8 on page 25 is discoverable as part of the multicast group `default`.

Configuring a client

To use the agent a client must be configured to connect to a broker using a discovery protocol that uses a multicast agent URI as shown in Example 2.9 on page 26.

Example 2.9. Client Connection URL using Multicast Discovery

```
discovery://(multicast://default)
```

A client using the URL in Example 2.9 on page 26 will discover all the brokers advertised in the `default` multicast group and generate a list of brokers to which it can connect.

Zeroconf Discovery Agent

Overview

The *zeroconf discovery agent* is derived from Apple's Bonjour Networking[1] technology, which defines the zeroconf protocol as a mechanism for discovering services on a network. Fuse Message Broker bases its implementation of the zeroconf discovery agent on JmDSN[2], which is a service discovery protocol that is layered over IP/multicast and is compatible with Apple Bonjour.

The agent requires that *each* broker you want to advertise is configured to use a multicast discovery agent to publish its details to a multicast group. Clients using the zeroconf agent as part of the discovery URI they use for connecting to a broker will use the agent to receive the list of available brokers advertising in the specified multicast group.

Important

Your local network (LAN) must be configured to use IP/multicast for the zeroconf agent to work.

URI

The zeroconf discovery agent URI conforms to the syntax in Example 2.10 on page 27.

Example 2.10. Zeroconf Discovery Agent URI Format

```
zeroconf://GroupID
```

Where the *`GroupID`* is an alphanumeric identifier. All participants in the same discovery network must use the same *`GroupID`*.

Configuring a broker

For a broker to be discoverable using the zeroconf discovery agent, you must enable a multicast discovery agent in the broker's configuration. To enable the multicast discovery agent you set the `transportConnector` element's `discoveryUri` attribute to a mulitcast discovery agent URI as shown in Example 2.11 on page 27.

Example 2.11. Enabling a Multicast Discovery Agent on a Broker

```
<transportConnectors>
  <transportConnector name="openwire"
```

[1] http://developer.apple.com/networking/bonjour/
[2] http://sourceforge.net/projects/jmdns/

```
    uri="tcp://localhost:61716"
    discoveryUri="multicast://NEGroup" />
</transportConnectors>
```

The broker configured in Example 2.11 on page 27 is discoverable as part of the multicast group `NEGroup`.

Configuring a client

To use the agent a client must be configured to connect to a broker using a discovery protocol that uses a zeroconf agent URI as shown in Example 2.12 on page 28.

Example 2.12. Client Connection URL using Zeroconf Discovery

```
discovery://(zeroconf://NEGroup)
```

A client using the URL in Example 2.12 on page 28 will discover all the brokers advertised in the `NEGroup` multicast group and generate a list of brokers to which it can connect.

Dynamic Discovery Protocol

Overview

The *dynamic discovery protocol* combines reconnect logic with a discovery agent to dynamically create a list of brokers to which the client can connect. The discovery protocol invokes a discovery agent in order to build up a list of broker URIs. The protocol then randomly chooses a URI from the list and attempts to establish a connection to it. If it does not succeed, or if the connection subsequently fails, a new connection is established to one of the other URIs in the list.

URI syntax

Example 2.13 on page 29 shows the syntax for a discovery URI.

Example 2.13. Dynamic Discovery URI

```
discovery://(DiscoveryAgentUri)?Options
```

`DiscoveryAgentUri` is URI for the discovery agent used to build up the list of available brokers. Discovery agents are described in "Discovery Agents" on page 23.

The options, *`?Options`*, are specified in the form of a query list. The discovery options are described in Table 2.3 on page 29. You can also inject transport options as described in "Setting options on the discovered transports" on page 30.

Tip

If no options are required, you can drop the parentheses from the URI. The resulting URI would take the form `discovery://DiscoveryAgentUri`

Transport options

The discovery protocol supports the options described in Table 2.3 on page 29.

Table 2.3. Dynamic Discovery Protocol Options

Option	Default	Description
`initialReconnectDelay`	`10`	Specifies, in milliseconds, how long to wait before the first reconnect attempt.

Option	Default	Description
`maxReconnectDelay`	`30000`	Specifies, in milliseconds, the maximum amount of time to wait between reconnect attempts.
`useExponentialBackOff`	`true`	Specifies if an exponential back-off is used between reconnect attempts.
`backOffMultiplier`	`2`	Specifies the exponent used in the exponential back-off algorithm.
`maxReconnectAttempts`	`0`	Specifies the maximum number of reconnect attempts before an error is sent back to the client. `0` specifies unlimited attempts.

Sample URI

Example 2.14 on page 30 shows a discovery URI that uses a multicast discovery agent.

Example 2.14. Discovery Protocol URI

```
discovery://(multicast://default)?initialReconnectDelay=100
```

Setting options on the discovered transports

The list of transport options, *`Options`*, in the discovery URI can also be used to set options on the *discovered* transports. If you set an option *not* listed in "Setting options on the discovered transports" on page 30, the URI parser attempts to inject the option setting into every one of the discovered endpoints.

Example 2.15 on page 30 shows a discovery URI that sets the TCP `connectionTimeout` option to 10 seconds.

Example 2.15. Injecting Transport Options into a Discovered Transport

```
discovery://(multicast://default)?connectionTimeout=10000
```

The 10 second timeout setting is injected into every discovered TCP endpoint.

Chapter 3. Master/Slave

Persistent messages require an additional layer of fault tolerance. In case of a broker failure, persistent messages require that the replacement broker has a copy of all the undelivered messages. Master/slave groups address this requirement by having a standby broker that either mirrors the active broker's persistence store or shares the active broker's data store.

A master/slave group consists of two or more brokers where one master broker is active and one or more slave brokers are on hot standby, ready to take over whenever the master fails or shuts down. All of the brokers store the message and event data processed by the master broker. So, when one of the slaves takes over as the new master broker the integrity of the messaging system is guaranteed.

Fuse Message Broker supports three master/slave broker configurations:

- Shared nothing—the master forwards a copy of every persistent message it receives to a single slave broker
- Shared file system—the master and the slaves use a common persistence store that is located on a shared file system
- Shared JDBC database—the masters and the slaves use a common JDBC persistence store

Shared Nothing Master/Slave

Overview

A shared nothing master/slave group replicates data between a pair of brokers using a dedicated connection. The advantage of this approach is that it does not require a shared database or a shared file system and thus does not have a single point of failure.

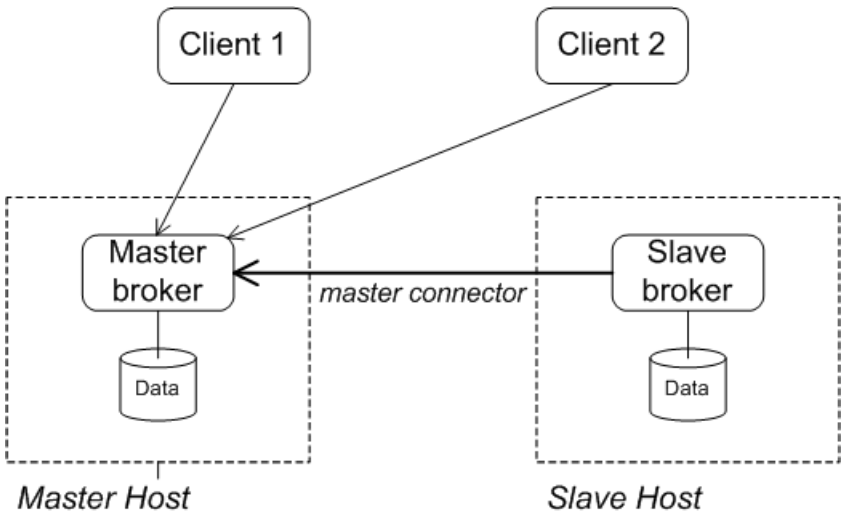
The disadvantage of this approach are:

- Reintroducing a failed master requires manually synchronizing the persistence stores and restarting the entire cluster.
- Persistent messaging suffers additional latency because producers must wait for messages to be replicated to the slave and be stored in the slave's persistent store

Initial state

Figure 3.1 on page 32 shows the initial state of a shared nothing master/slave group.

Figure 3.1. Shared Nothing Master/Slave Group Initial State



In this topology, the master broker does not require any special configuration. Unless specifically configured to wait for a slave, the master broker functions like an ordinary broker until a slave broker connects to it. Once

a slave connects to the master broker, the master broker forwards all events to the slave. It will not respond to a client request until the associated event has been successfully forwarded.

The slave broker is configured with a *master connector*, which connects to the master broker in order to duplicate the data stored in the master. While the connection is active, the slave consumes all events from the master: including messages, acknowledgments, and transactional states. The slave does *not* start any transport connectors or network connectors. Its sole purpose is to duplicate the state of the master.

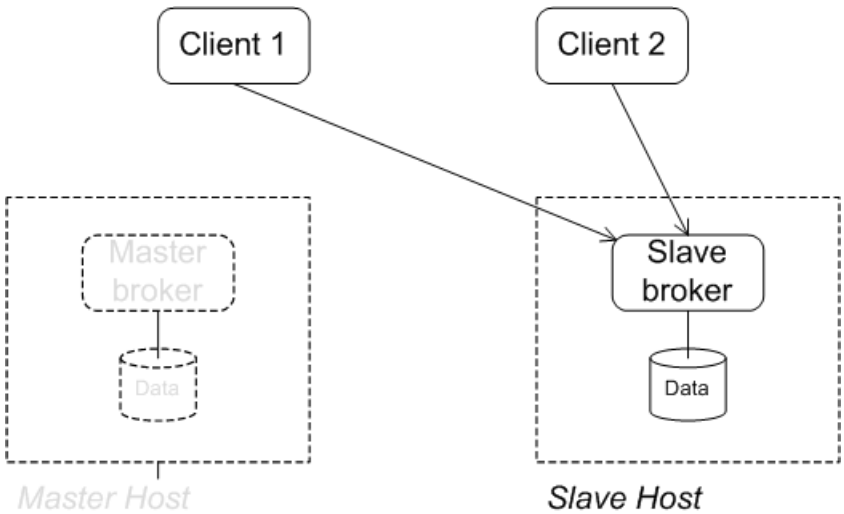
State after failure of the master

When the master fails, the slave can be configured to behave in one of two ways:

- *take over*—the slave starts up all of its transport connectors and network connectors and takes the place of the master broker. Clients that are configured to fail over experience no down time.
- *close down*—the slave stays unreachable and the client connections are shutdown until the master is reintroduced. The slave's data store is used as a back-up for the master in the case of a catastrophic hardware failure.

Figure 3.2 on page 33 shows the state of the master/slave group after the master broker has failed and the slave broker has taken over from the master.

Figure 3.2. Shared Nothing Master/Slave Group after Master Failure



Configuring the master

In a shared nothing master/slave group the master broker does not require any special configuration. When a slave broker opens a master connector to a broker, the broker is automatically turned into a master.

There are optional attributes you can set on the master's `broker` element that controls how the master behaves in relation to a slave broker. Table 3.1 on page 34 describes these attributes.

Table 3.1. Configuration Options for a Master in a Shared Nothing Master/Slave Group

Attribute	Default	Description
`waitForSlave`	`false`	Specifies if the master will wait for a slave to connect before it will accept client connections.
`shutdownOnSlaveFailure`	`false`	Specifies if the master will stop processing client requests if it loses the connection to the slave broker.

Example 3.1 on page 34 shows a sample configuration for a master broker in a shared nothing master/slave group.

Example 3.1. Master Configuration for Shared Nothing Master/Slave Group

```
<broker brokerName="master"
        waitForSlave="true"
        shutdownOnSlaveFailure="false"
        xmlns="http://activemq.apache.org/schema/core">
  ...
  <transportConnectors>
     <transportConnector uri="tcp://masterhost:61616"/>
  </transportConnectors>
  ...
</broker>
```

Important

You should **not** configure a network connector between the master and its slave. If you configure a network connector, you may encounter race conditions when the master broker is under heavy load.

Configuring the slave

When using shared nothing master/slave there are two approaches to configuring the slave:

- Configure the master connector as a broker service.

  In this approach you configure the master connector by adding a `masterConnector` child to the broker's `services` element.

  The advantage of this approach is that it allows you to provide user credentials to a secure master broker. The disadvantage is that you cannot configure the slave to shutdown on master failure. It will always takeover the master role.

  The `masterConnector` element has three attributes, described in Table 3.2 on page 35, that are used to configure the connector.

Table 3.2. Attributes for Configuring the Master Connector Service

Attribute	Description
`remoteURI`	Specifies the master's transport connector that will be used by the master connector.
`userName`	Specifies the user name used to connect to the master.
`password`	Specifies the password used to connect to the master.

Example 3.2 on page 35 shows how to configure the slave using the `masterConnector` element.

Example 3.2. Configuring the Master Connector as a Service

```
<broker brokerName="slave"
        xmlns="http://activemq.apache.org/schema/core">
  ...
  <services>
    <masterConnector
      remoteURI="tcp://localhost:62001"
      userName="James"
      password="Cheese" />
  </services>

  <transportConnectors>
    <transportConnector uri="tcp://slavehost:61616"/>
  </transportConnectors>
  ...
</broker>
```

- Configure the master connector directly on the broker.

  In this approach you configure the master connector by setting the attributes described in Table 3.3 on page 36 directly on the `broker` element.

  Table 3.3. Attributes for Configuring a Master Connector on the Broker

Attribute	Description
`masterConnectorURI`	Specifies the master's transport connector that will be used by the master connector.
`shutdownOnMasterFailure`	Specifies if the slave shuts down when it loses the connection to the master.

  The advantage of this approach is that you can configure the slave to simply serve as a back-up for the master broker and shut down when the master shuts down. The disadvantage is that you cannot connect to masters that require authentication.

  Example 3.3 on page 36 shows how to configure the master connector by setting attributes on the `broker` element.

  Example 3.3. Configuring the Master Connector Directly

```
<broker brokerName="slave"
        masterConnectorURI="tcp://masterhost:62001"
        shutdownOnMasterFailure="false"
        xmlns="http://activemq.apache.org/schema/core">
  ...
  <transportConnectors>
     <transportConnector uri="tcp://slavehost:61616"/>
  </transportConnectors>
  ...
</broker>
```

Configuring the clients

Assuming that you choose the mode of operation where the slave takes over from the master, your clients will need to include logic for failing over to the new master. Adding the fail over logic requires that the clients use the masterslave protocol. This protocol is an instance of the failover protocol described in "Failover Protocol" on page 14 that is specifically tailored for shared noting master/slave pairs.

If you had a two broker cluster where the master is configured to accept client connections on `tcp://masterhost:61616` and the slave is configured accept client connections on `tcp://slavehost:61616`, you would use the masterslave URI shown in Example 3.4 on page 37 for your clients.

Example 3.4. URI for Connecting to a Master/Slave Cluster

```
masterslave://(tcp://masterhost:61616,tcp://slavehost:61616)
```

Reintroducing the master

Reintroducing the master broker after a failure is a manual process. Perform the following steps:

1. Shut down the slave.
2. Copy the slave's data directory over to the master's data directory.
3. Start the master and the slave.

Shared File System Master/Slave

Overview

A shared file system master/slave group works by sharing a common data store that is located on a shared file system. Brokers automatically configure themselves to operate in master mode or slave mode based on their ability to grab an exclusive lock on the underlying data store.

The advantage of this configuration are:

- that a group can consist of more than two members. I group can have an arbitrary number of slaves.
- that a failed node can rejoin the group without any manual intervention. When a new node joins, or rejoins, the group it automatically falls into slave mode until it can get an exclusive lock on the data store.

The disadvantage of this configuration is that the shared file system is a single point of failure. This disadvantage can be mitigated by using a storage area network(SAN) with built in high availability(HA) functionality. The SAN will handle replication and fail over of the data store.

File locking requirements

The shared file system requires an efficient and reliable file locking mechanism to function correctly. Not all SAN file systems are compatible with the shared file system configuration's needs.

Warning

OCFS2 is incompatible with this master/slave configuration, because mutex file locking from Java is not supported.

Warning

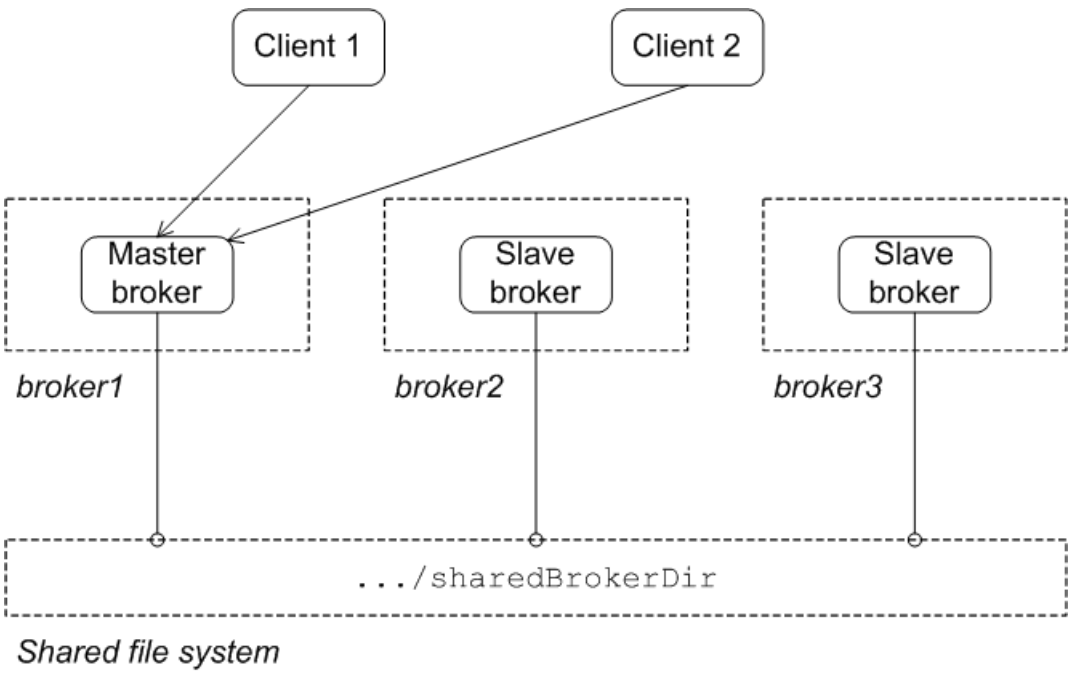
NFSv3 is incompatible with this master/slave configuration. In the event of an abnormal termination of a master broker, which is an NFSv3 client, the NFSv3 server does not time out the lock held by the client. This renders the Fuse Message Broker data directory inaccessible. Because of this, the slave broker cannot acquire the lock and therefore cannot start up. In this case, the only way to unblock the master/slave in NFSv3 is to reboot all broker instances.

On the other hand, NFSv4 *is* compatible with this master/slave configuration, because its design includes timeouts for locks. When an NFSv4 client holding a lock terminates abnormally, the lock is automatically released after 30 seconds, allowing another NFSv4 client to grab the lock.

Initial state

Figure 3.3 on page 39 shows the initial state of a shared file system master/slave group. When all of the brokers are started, one of them grabs the exclusive lock on the broker data store and becomes the master. All of the other brokers remain slaves and pause while waiting for the exclusive lock to be freed up. Only the master starts its transport connectors, so all of the clients connect to it.

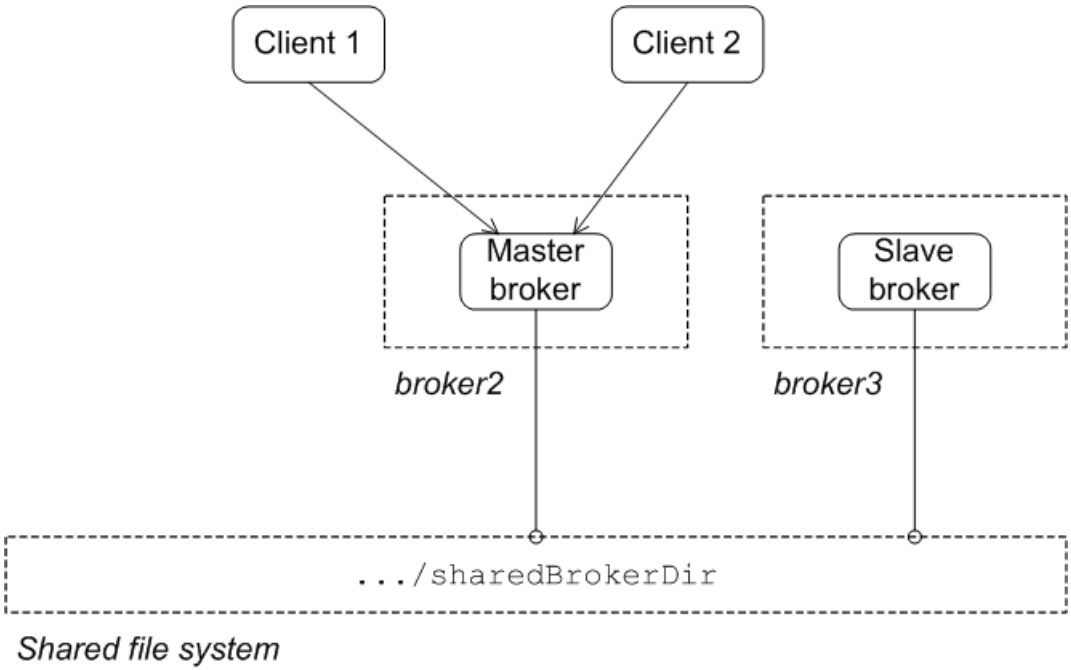
Figure 3.3. Shared File System Initial State



State after failure of the master

Figure 3.4 on page 40 shows the state of the master/slave group after the original master has shut down or failed. As soon as the master gives up the lock (or after a suitable timeout, if the master crashes), the lock on the data store frees up and another broker grabs the lock and gets promoted to master.

Figure 3.4. Shared File System after Master Failure



After the clients lose their connection to the original master, they automatically try all of the other brokers listed in the failover URL. This enables them to find and connect to the new master.

Configuring the brokers

In the shared file system master/slave configuration, there is nothing special to distinguish a master broker from the slave brokers. The membership of a particular master/slave group is defined by the fact that all of the brokers in the group use the *same* persistence layer and store their data in the *same* shared directory.

Example 3.5 on page 40 shows the broker configuration for a shared file system master/slave group that shares a data store located at `/sharedFileSystem/sharedBrokerData` and uses the KahaDB persistence store.

Example 3.5. Shared File System Broker Configuration

```
<broker ... >
  ...
  <persistenceAdapter>
    <kahaDB directory="/sharedFileSystem/sharedBrokerData"/>
```

```
  </persistenceAdapter>
  ...
</broker>
```

All of the brokers in the group *must* share the same `persistenceAdapter` element.

Configuring the clients

Clients of shared file system master/slave group must be configured with a failover URL that lists the URLs for all of the brokers in the group. Example 3.6 on page 41 shows the client failover URL for a group that consists of three brokers: `broker1`, `broker2`, and `broker3`.

Example 3.6. Client URL for a Shared File System Master/Slave Group

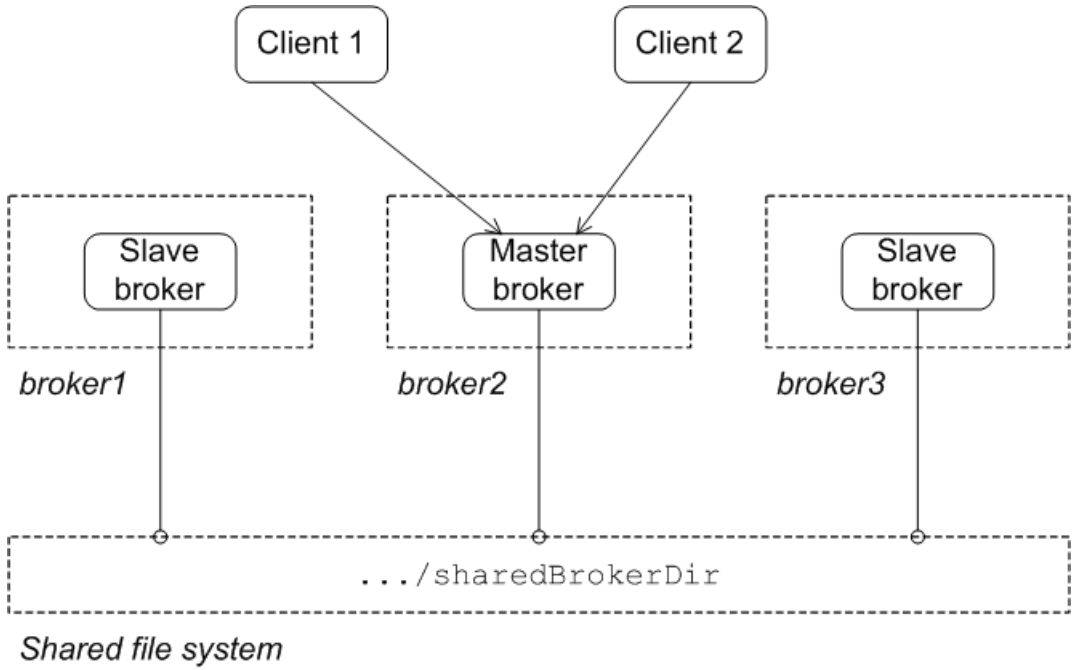
```
failover:(tcp://broker1:61616,tcp://broker2:61616,tcp://broker3:61616)
```

For more information about using the failover protocol see "Static Failover" on page 15.

Reintroducing a failed node

You can restart the failed master at any time and it will rejoin the cluster. It will rejoin as a slave broker because one of the other brokers already owns the exclusive lock on the data store, as shown in Figure 3.5 on page 42.

Figure 3.5. Shared File System after Master Restart



Shared JDBC Master/Slave

Overview

A shared JDBC master/slave group works by sharing a common database using the JDBC persistence adapter. Brokers automatically configure themselves to operate in master mode or slave mode, depending on whether or not they manage to grab a mutex lock on the underlying database table.

The advantage of this configuration are:

- that a group can consist of more than two members. I group can have an arbitrary number of slaves.
- that a failed node can rejoin the group without any manual intervention. When a new node joins, or rejoins, the group it automatically falls into slave mode until it can get a mutex lock on the database table.

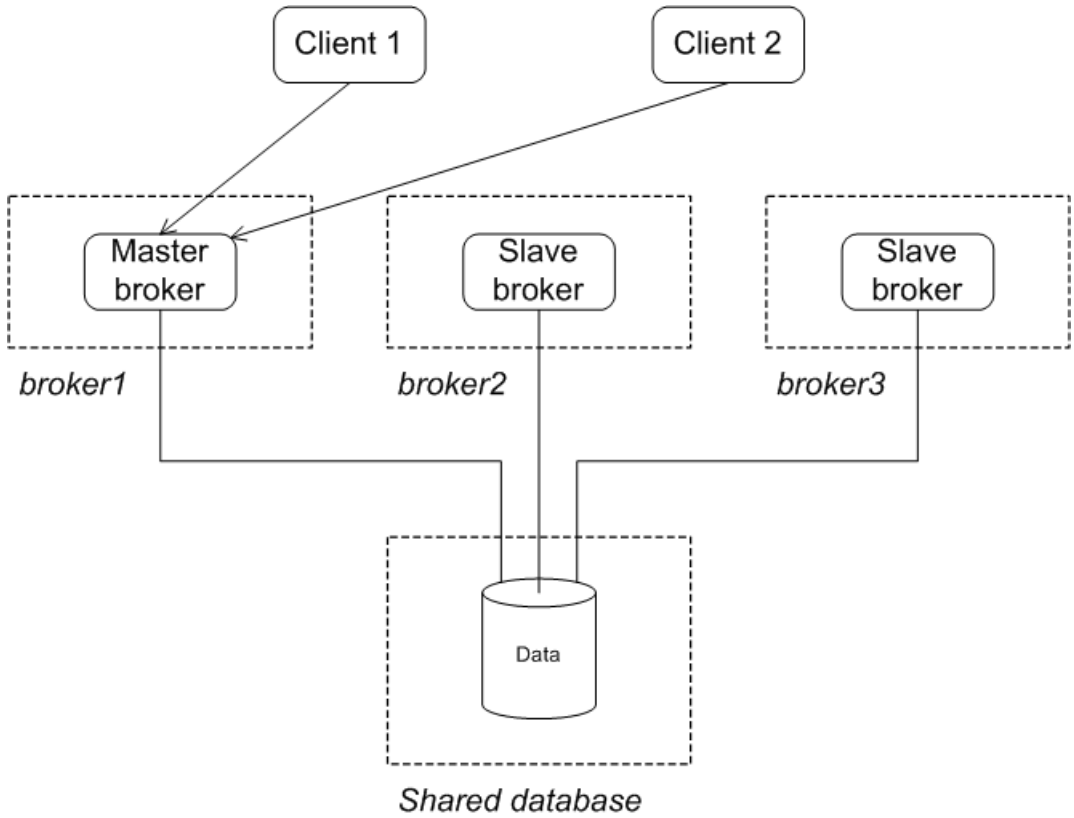
The disadvantages of this configuration are:

- The shared database is a single point of failure. This disadvantage can be mitigated by using a database with built in high availability(HA) functionality. The database will handle replication and fail over of the data store.
- You cannot enable high speed journaling. This has a significant impact on performance.

Initial state

Figure 3.6 on page 44 shows the initial state of a JDBC master/slave group. When all of the brokers are started, one of them grabs the mutex lock on the database table and becomes the master. All of the other brokers become slaves and pause while waiting for the lock to be freed up. Only the master starts its transport connectors, so all of the clients connect to it.

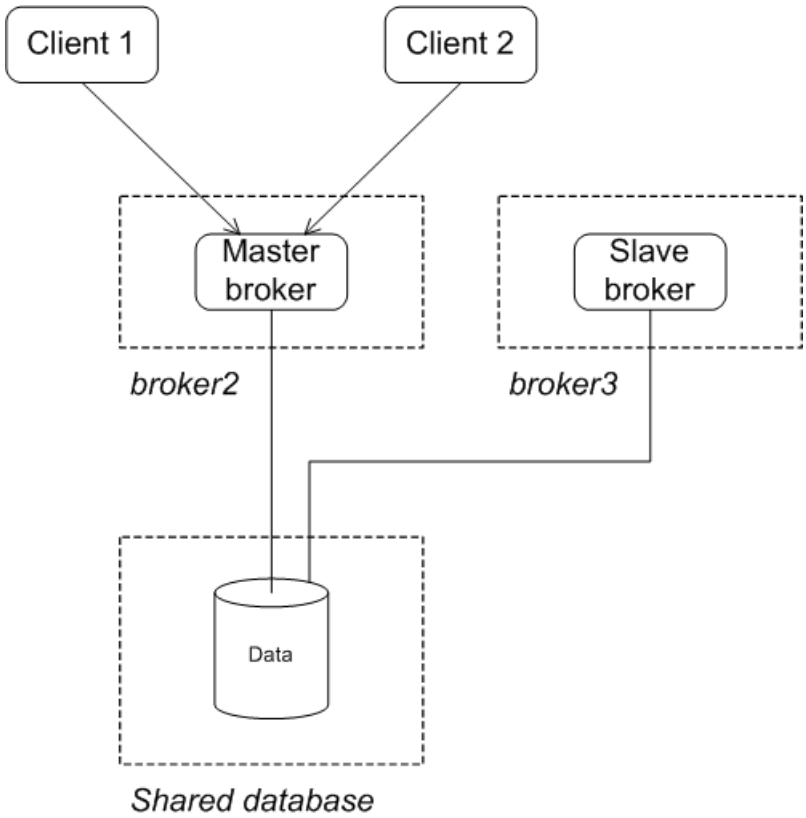
Figure 3.6. JDBC Master/Slave Initial State



After failure of the master

Figure 3.7 on page 45 shows the state of the group after the original master has shut down or failed. As soon as the master gives up the lock (or after a suitable timeout, if the master crashes), the lock on the database table frees up and another broker grabs the lock and gets promoted to master.

Figure 3.7. JDBC Master/Slave after Master Failure



After the clients lose their connection to the original master, they automatically try all of the other brokers listed in the failover URL. This enables them to find and connect to the new master.

Configuring the brokers

In a JDBC master/slave configuration, there is nothing special to distinguish a master broker from the slave brokers. The membership of a particular master/slave group is defined by the fact that all of the brokers in the cluster use the *same* JDBC persistence layer and store their data in the *same* database tables.

Example 3.7 on page 46 shows the configuration used be a master/slave group that stores the shared broker data in an Oracle database.

Example 3.7. JDBC Master/Slave Broker Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:amq="http://activemq.apache.org/schema/core"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
                      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
  http://activemq.apache.org/schema/core
  http://activemq.apache.org/schema/core/activemq-core-5.3.1.xsd">

    <broker xmlns="http://activemq.apache.org/schema/core"
            brokerName="brokerA">
        ...
        <persistenceAdapter>
            <jdbcPersistenceAdapter dataSource="#oracle-ds"/>
        </persistenceAdapter>
        ...
    </broker>

    <bean id="oracle-ds"
          class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@localhost:1521:AMQDB"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
        <property name="poolPreparedStatements" value="true"/>
    </bean>

</beans>
```

Important

The persistence adapter is configured as a direct JDBC persistence layer, using the `jdbcPersistenceAdapter` element. You must *not* use the journaled persistence adapter, which is configured using the `journalPersistenceAdapter` element, in this scenario.

Configuring the clients

Clients of shared JDBC master/slave group must be configured with a failover URL that lists the URLs for all of the brokers in the group. Example 3.8 on page 47 shows the client failover URL for a group that consists of three brokers: `broker1`, `broker2`, and `broker3`.

Example 3.8. Client URL for a Shared JDBC Master/Slave Group

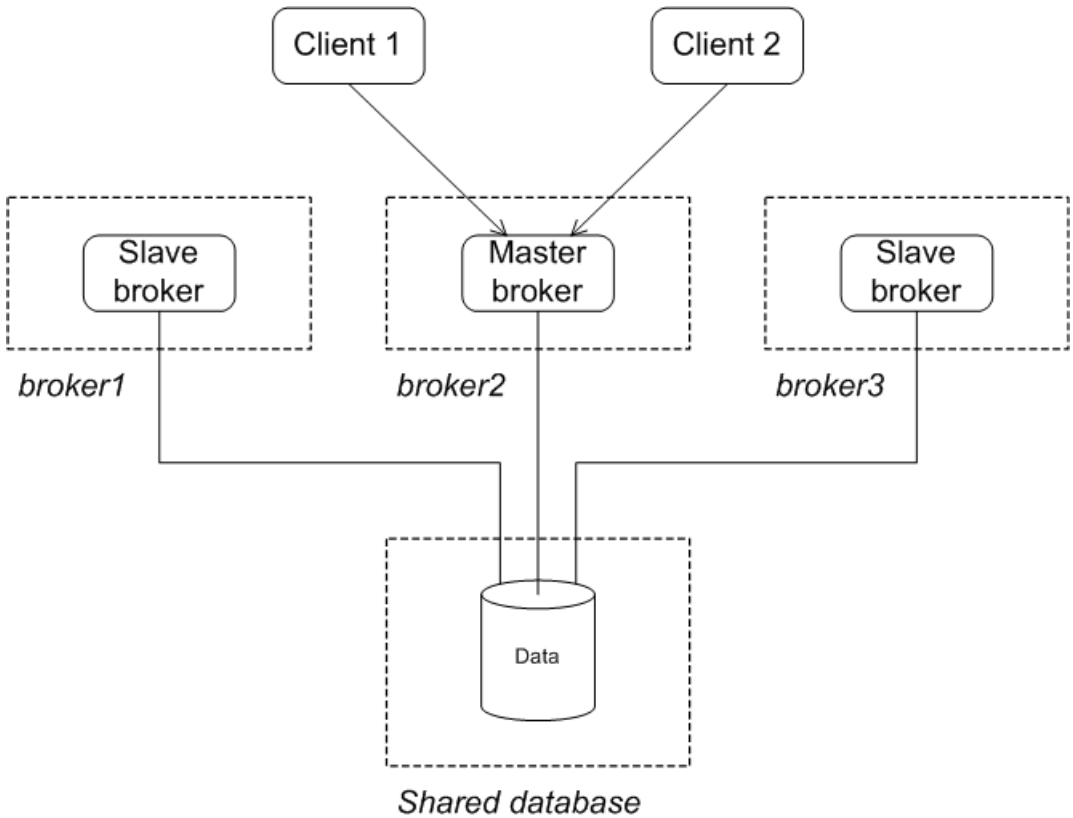
```
failover:(tcp://broker1:61616,tcp://broker2:61616,tcp://broker3:61616)
```

For more information about using the failover protocol see "Static Failover" on page 15.

Reintroducing a failed node

You can restart the failed node at any time and it will rejoin the group. It will rejoin the group as a slave because one of the other brokers already owns the mutex lock on the database table, as shown in Figure 3.8 on page 47.

Figure 3.8. JDBC Master/Slave after Master Restart



Chapter 4. Master/Slave and Broker Networks

Master/slave groups and networks of brokers are very different things. Master/slave groups can be used in a network of brokers to provide fault tolerance to the nodes in the broker network. This requires careful consideration and the use of a special network connection protocol.

Overview

Master/slave groups and broker networks represent different levels of organization. You can include a master/slave group as a node in a network of brokers. Using the basic principles of making a master/slave group a node in a broker network, you can scale up to an entire network consisting of master/slave groups.

When combining master/slave groups with broker networks there are two things to remember:

- Network connectors to a master/slave group use a special protocol.
- A broker cannot open a network connection to another member of its master/slave group.

Configuring the connection to a master/slave group

The network connection to a master/slave group needs to do two things:

- Open a connection to the master broker without connecting to the slave brokers.
- Connect to the new master in the case of a failure.

The network connector's reconnect logic will handle the reconnection to the new master in the case of a network failure. The network connector's connection logic, however, attempts to establish connections to all of the specified brokers. To get around the network connector's default behavior, you use a masterslave URI to specify the list of broker's in the master/slave group. The masterslave URI only allows the connector to connect to one of brokers in the list which will be the master.

The masterslave protocol's URI is a list of the connections points for each broker in the master/slave group. The network connector will traverse the list in order until it establishes a connection.

Example 4.1 on page 50 shows a network connector configured to link to a master/slave group.

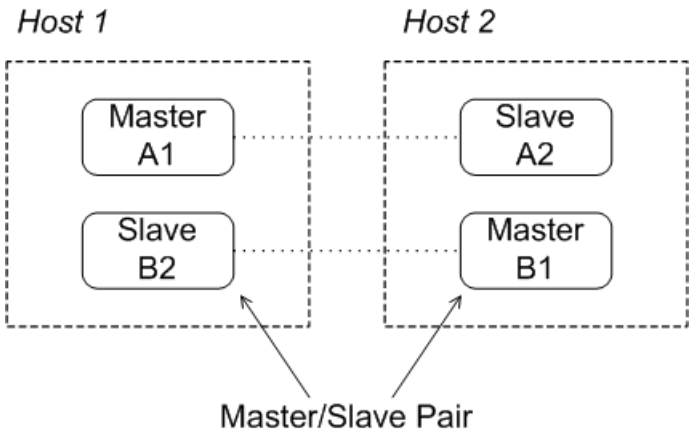
Example 4.1. Network Connector to a Master/Slave Group

```
<networkConnectors>
  <networkConnector name="linkToCluster"
                uri="mastersalve:(tcp://masterHost:61002,tcp://slaveHost:61002)"
                ... />
</networkConnectors>
```

Host pair with master/slave groups

In order to scale up to a large fault tolerant broker network, it is a good idea to adopt a simple building block as the basis for the network. An effective building block for this purpose is the host pair arrangement shown in Figure 4.1 on page 50.

Figure 4.1. Master/Slave Groups on Two Host Machines



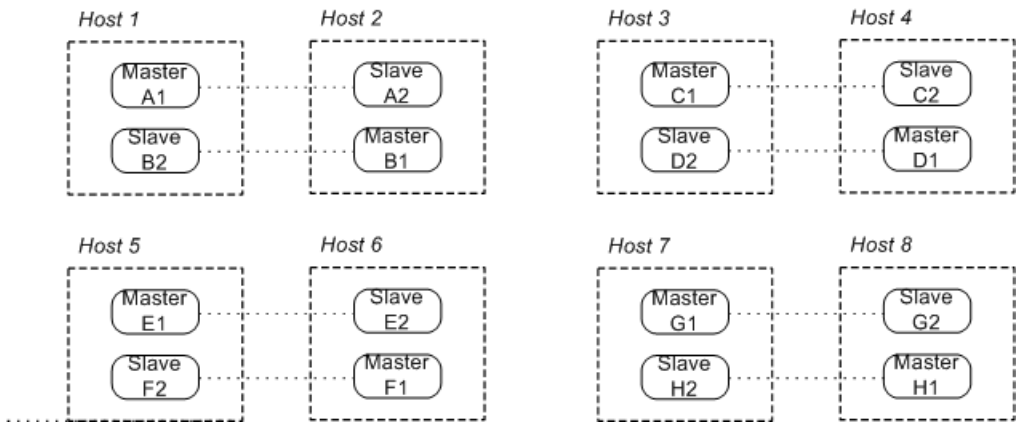
The host pair arrangement consists of two master/slave groups distributed between two host machines. Under normal operating conditions, one master broker is active on each of the two host machines. If one of the machines should fail for some reason, the slave on the other machine takes over, so that you end up with two active brokers on the healthy machine.

When configuring the network connectors, you must remember **not** to open any connectors to brokers in the same group. For example, the network connector for `brokerB1` should be configured to connect to at most `brokerA1` and `brokerA2`.

Network of multiple host pairs

You can easily scale up to a large fault tolerant broker network by adding host pairs, as shown in Figure 4.2 on page 51.

Figure 4.2. Broker Network Consisting of Host Pairs



The preceding network consists of eight master/slave groups distributed over eight host machines. As before, you should open network connectors only to brokers outside the current master/slave group. For example, `brokerA1` can connect to at most the following brokers: `brokerB*`, `brokerC*`, `brokerD*`, `brokerE*`, `brokerF*`, `brokerG*`, and `brokerH*`.

More information

For detailed information on setting up a network of brokers see *Using Networks of Brokers*.

Index