Fuse ESB

Clustering Guide

Version 4.4.1
Sept. 2011

Clustering Guide

Version 4.4.1

Updated: 06 Jun 2013

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. Failover and Discovery Protocols

Failover and discovery protocols play a key role in broker networks and clusters. When a client connects to a broker network, it needs a mechanism to discover the brokers in the network and, in the event that the client-broker connection fails, the client must be able to fail over to another broker instance.

Failover Protocol

Overview

The *failover protocol* is primarily used by clients to facilitate 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 transport connectors. 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.

Brokers in a network of brokers can be configured to dynamically update their clients' failover lists. The brokers inform their clients about the other brokers in the network so that in the event of a failure the clients know how to connect to a running broker. The brokers update their clients as new brokers join and leave the network.

For more information about configuring a client to use the failover protocol see "Failover Protocol" in *Broker Client Connectivity Guide*.

Using failover in the broker

Brokers should never use the failover protocol when configuring its transport connectors. The failover protocol does not support listening for incoming connections.

In general, brokers should not use the failover protocol when configuring its network connectors. The failover protocol's reconnect logic cannot recreate the network bridge between two brokers. It interferes with the network connectors reconnect logic which is required to rebuild dropped network bridges.

The one case where you want to use the failover protocol in a network connector is when you are attempting to create a network of brokers that includes a master/slave cluster. See "Fault Tolerant Broker Network" on page 81 for more information.

Configuring the broker to participate in dynamic failover

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 "Broker Networks" on page 23 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 1.1 on page 13 shows the broker properties that configure a failover cluster. These properties are exposed as attributes on the `transportConnector` element.

Table 1.1. Transport Connector 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 use for reconnect in the case of failure.

Note

The update and rebalance features should only be enabled, only if the clients that connect to the broker cluster use Fuse Message Broker 5.4.0 or later. These features are incompatible with clients older than version 5.4.0.

Example

Example 1.1 on page 13 shows the configuration for a broker that participates in dynamic failover.

Example 1.1. 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 1.1 on page 13 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 1.2 on page 14 shows the URI for a client that uses the failover protocol to connect to the broker and its cluster.

Example 1.2. Failover URI for Connecting to a Failover Cluster

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

Dynamic Discovery Protocol

Overview

The *dynamic discovery protocol* combines reconnect logic with the capability to auto-discover broker endpoints in the local network. 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 it subsequently fails, a new connection is established to one of the other URIs in the list.

Discovery agents

A *discovery agent* is a bootstrap mechanism that enables a message broker, consumer, or producer to obtain a list of broker URIs, where the URIs represent connector endpoints. The broker, consumer, or producer can subsequently connect to one of the URIs in the list.

The following kinds of discovery agent are currently supported in Fuse Message Broker:

- Simple (static) discovery agent.
- Multicast discovery agent.
- Rendezvous discovery agent.

For more details, see "Discovery Agents" on page 18.

Configuring a transport connector with a discovery agent

Before you can use the discovery protocol, you must make your broker's endpoints discoverable by adding a discovery agent to each transport connector. The static discovery mechanism is a special case however: you do *not* need to add a discovery agent to the broker endpoints in order to use a static discovery agent on the client side.

For example, to make a TCP transport connector discoverable, set the `discoveryUri` attribute on the `transportConnector` element as follows:

Example 1.3. Making a TCP connector discoverable

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

Where the TCP transport connector is configured to use the multicast discovery agent, `multicast://default`.

Configuration syntax

A `discovery` URI must conform to the following syntax:

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

Where the discovery agent URI, *`DiscoveryAgentUri`*, identifies a discovery agent, as described in "Discovery agents" on page 15 above. The transport options, *`?TransportOptions`*, are specified in the form of a query list (where the supported options are described in Table 1.2 on page 16). If no transport options are required, you can use the following alternative syntax:

```
discovery://DiscoveryAgentUri
```

Transport options

The discovery protocol supports the transport options described in Table 1.2 on page 16

Table 1.2. Discovery Transport Options

Option Name	Default	Description
`initialReconnectDelay`	`10`	How long to wait before the first reconnect attempt (in ms).
`reconnectDelay`	`10`	How long to wait for discovery.
`maxReconnectDelay`	`30000`	The maximum amount of time to wait between reconnect attempts (in ms).
`useExponentialBackOff`	`true`	If `true`, use an exponential back-off between reconnect attempts.
`backOffMultiplier`	`2`	The exponent used in the exponential back-off algorithm.
`maxReconnectAttempts`	`0`	If not `0`, this is the maximum number of reconnect attempts before an error is sent back to the client.
`group`	`default`	A group identifier used to partition multicast traffic among collaborating peers; the group forms part of the shared identity of a discovery datagram.

Applying options to the discovered transport

It is also possible to apply options to the discovered transport by adding those options to the discovery URI. When parsing the URI, the discovery protocol consumes those options that it recognizes (the ones listed in Table 1.2 on page 16). Any remaining options are held in reserve until a transport is discovered. When the discovered transport is resolved, the remaining options are applied to that transport.

For example, suppose that you expect to discover a TCP endpoint, you can then add some TCP transport options to your discovery URI. The following discovery URI shows how to set the `connectionTimeout` TCP transport option:

```
discovery://(multicast://default)?connectionTimeout=2000
```

Sample URI

The following is an example of a discovery URI that uses a multicast discovery agent:

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

Discovery Agents

Overview

A discovery agent is a bootstrap mechanism that enables a client or message broker to discover other broker instances on a network. A discover agent URI is used on the client side and on the broker side, as follows:

- *Client side*—the discovery agent URI resolves to a list of broker URIs. To use a discovery agent URI on the client side, you must insert it into a dynamic discovery URI, `discovery://(...)`, which then opens a connection to one of the URIs in the list.
- *Broker side*—in order to make a broker discoverable, it is usually necessary to configure a discovery agent in the broker as well (an exception to this requirement is the `simple` discovery agent).

Note

A discover agent URI resolves to a list of transport URIs, but *the discovery agent URI is not itself a transport URI and cannot be used in place of a transport URI*.

Configuring a discovery agent on the client side

Since a discovery agent is not a transport protocol, you cannot use a discovery agent URI directly on the client side. To use a discovery agent on the client side, embed the agent URI,*`DiscoveryAgentUri`*, inside a discovery URL, as follows:

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

The client recognizes the discovery URL as a transport. It first obtains a list of available endpoint URLs using the specified discovery agent and then connects to one of the discovered URLs. For more details about the discovery protocol, see "Dynamic Discovery Protocol" on page 15 .

Configuring discovery agents on a message broker

For certain kinds of discovery agent (for example, multicast or rendezvous), 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, edit the relevant `transportConnector` element as follows:

```
<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`. You can associate multiple endpoints with the same discovery agent. For example, to configure both an Openwire endpoint and a Stomp endpoint to use the `multicast://default` discovery agent:

```
<transportConnectors>
  <transportConnector
            name="openwire"
            uri="tcp://localhost:61716"
            discoveryUri="multicast://default"/>
  <transportConnector
            name="stomp"
            uri="stomp://localhost:61613"
            discoveryUri="multicast://default"/>
</transportConnectors>
```

Discovery agents

Fuse Message Broker currently supports the following discovery agents:

- "Simple (static) discovery agent"
- "Multicast discovery agent"
- "Rendezvous discovery agent"

Simple (static) discovery agent

The simple (static) discovery agent provides an explicit list of broker URLs for a client to connect to. For example:

```
simple://(tcp://localhost:61716,tcp://localhost:61816)
```

In general, the URI for a simple discovery agent must conform to the following syntax:

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

Or equivalently:

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

The two prefixes, `simple:` and `static:`, are exactly equivalent. In order to use the agent URI, it *must* be embedded inside a discovery URL—for example:

Example 1.4. Simple discovery agent URL

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

This discovery agent is only used on the client side. No extra configuration is required in the broker.

Multicast discovery agent

The *multicast discovery agent* uses the IP multicast protocol to find any message brokers currently active on the local network. In order for the protocol to work, a multicast discovery agent must be enabled on *each* broker you want to advertise and messaging clients must be configured to use a `discovery` URI.

The URI for a multicast discovery agent must conform to the following syntax:

```
multicast://Address:Port?TransportOptions
```

The following transport options are supported:

Table 1.3. Multicast Discovery Agent Options

Option Name	Default	Description
`group`	`default`	Specify a unique group name that partitions multicast traffic.
`minimumWireFormatVersion`	`0`	The minimum wireformat version that is allowed.
`trace`	`false`	Causes all commands sent over the transport to be logged.
`useLocalHost`	`true`	When `true`, causes the local machine name to resolve to `localhost`.
`datagramSize`	`4 * 1024`	Specifies the size of a datagram.
`timeToLive`	`-1`	The datagram time-to-live. Set greater than `1` in order to send packts beyond the local network. *Note:* For IPv4 addresses, you must also set the Java system property, `java.net.preferIPv4Stack=true`. See the IPv6 User Guide for JDK[1].
`loopBackMode`	`false`	Specifies whether or not to use loopback mode.

[1] http://java.sun.com/j2se/1.5.0/docs/guide/net/ipv6_guide/index.html

Option Name	Default	Description
`wireFormat`	`default`	The name of the wireformat to use.
`wireFormat.*`		All properties with this prefix are used to configure the wireformat.

All participants in the same discovery network must use the same group ID. To specify the IP adress, port number, and group ID explicitly, you can specify a URI like the following:

```
multicast://224.1.2.3:6255?group=mygroupname
```

For easy configuration, you could use the special default multicast URI, as follows:

```
multicast://default
```

This default URI is equivalent to the URI, `multicast://239.255.2.3:6155?group=default`.

Note

Your local network (LAN) must be configured appropriately for the IP/multicast protocol to work. If your clients fail to discover a broker using the `multicast` protocol, this could be because IP/multicast is not set up on your network.

Rendezvous discovery agent

The *rendezvous discovery agent* is derived from Apple's Bonjour Networking[2] technology, which defines the rendezvous protocol as a mechanism for discovering services on a network. To enable the protocol, a multicast discovery agent must be configured on *each* broker you want to advertise and messaging clients must be configured to use a `discovery` URI.

The URI for a rendezvous discovery agent must conform to the following syntax:

```
rendezvous://GroupID
```

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

For example, to use a rendezvous discovery agent on the client side, where the client needs to connect to the `groupA` group, you would construct a discovery URL like the following:

```
discovery://(rendezvous://groupA)
```

[2] http://developer.apple.com/networking/bonjour/

Note

Your local network (LAN) must be configured appropriately for the IP/multicast protocol to work. If your clients fail to discover a broker using the `rendezvous` protocol, this could be because IP/multicast is not set up on your network.

Chapter 2. Broker Networks

Fuse Message Broker has the capability to connect brokers together to form networks. In general, the main reason to form such networks of brokers is to achieve greater scalability, where broker-to-broker connections are used to consolidate message flow. It can also be a useful way to manage messaging domains, with certain kinds of message constrained to propagate only in one part of the broker network.

Network Connectors

Overview

Network connectors define the broker-to-broker links that are the basis of a broker network. This section defines the basic options for configuring network connectors and explains the concepts that underlie them.

Active consumers

An *active consumer* is a consumer that is connected to one of the brokers in the network, has indicated to the broker which topics and queues it wants to receive messages on, and is ready to receive messages. The broker network has the ability to keep track of active consumers, receiving notifications whenever a consumer connects to or disconnects from the network.

Subscriptions

In the context of a broker network, a *subscription* is a block of data that represents an active consumer's interest in receiving messages on a particular queue or on a particular topic. Brokers use the subscription data to decide what messages to send where. Subscriptions, therefore, encapsulate all of the information that a broker might need to route messages to a consumer, including JMS selectors and which route to take through the broker network.

Subscriptions are inherently dynamic. If a given consumer disconnects from the broker network (thus becoming inactive), its associated subscriptions are automatically cancelled throughout the network.

Note

This usage of the term, *subscription*, deviates from standard JMS terminology, where there can be topic subscriptions but there is no such thing as a queue subscription. In the context of broker networks, however, we speak of both *topic subscriptions* and *queue subscriptions*.

Propagation of subscriptions

Both topic subscriptions and queue subscriptions propagate automatically through a broker network. That is, when a consumer connects to a broker, it passes its subscriptions to the local broker and the local broker then forwards the subscriptions to neighbouring brokers. This process continues until the subscriptions are propagated throughout the broker network.

Under the hood, Fuse Message Broker implements subscription propagation using *advisory messages*, where an advisory message is a message sent through one of the special channels known as an *advisory topic*. An

advisory topic is essentially a reserved JMS topic used for transmitting administrative messages. All advisory topics have names that start with the prefix, `ActiveMQ.Advisory`.

Warning

In order for dynamic broker networks to function correctly, it is essential that advisory messages are enabled (which they are by default). Make sure that you do *not* disable advisory messages on any broker in the network. For example, if you are configuring your brokers using XML, make sure that the `advisorySupport` attribute on the `broker` element is *not* set to `false`.

In principle, it *is* possible to configure a static broker network when advisory messages are disabled. See "Dynamic and Static Propagation" on page 39 for details.

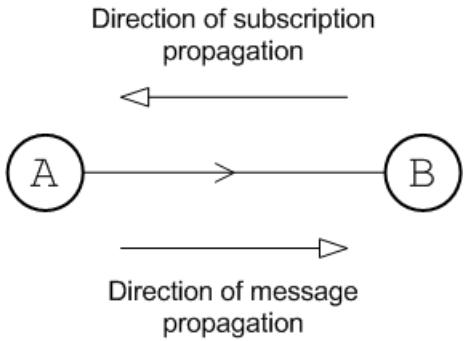
Network connector

A broker network is built up by defining directed connections from one broker to another, using *network connectors*. The broker that establishes the connection *passes messages* to the broker it is connected to. In XML, a network connector is defined using the `networkConnector` element, which is a child of the `networkConnectors` element.

Single connector

Figure 2.1 on page 26 shows a single network connector from broker A to broker B. The arrow on the connector indicates the direction of message propagation (from A to B). Subscriptions propagate in the *opposite* direction (from B to A). Because of the restriction on the direction of message flow in this network, it is advisable to connect producers only to broker A and consumers only to broker B. Otherwise, some messages might not be able to reach the intended consumers.

Figure 2.1. Single Connector



When the connector arrow points from A to B, this implies that the network connector is actually defined on broker A. For example, the following fragment from broker A's configuration file shows the network connector that connects to broker B:

Example 2.1. Single connector configuration

```
<beans ...>
    <broker xmlns="http://activemq.apache.org/schema/core"
            brokerName="brokerA" brokerId="A" ... >
        ...
        <networkConnectors>
            <networkConnector name="linkToBrokerB"
                uri="static:(tcp://localhost:61002)"
                networkTTL="3"
            />
        </networkConnectors>
        ...
        <transportConnectors>
            <transportConnector name="openwire" uri="tcp://0.0.0.0:61001"/>
        </transportConnectors>
    </broker>
</beans>
```

The `networkConnector` element in the preceding example sets the following basic attributes:

`name`

Identifies this network connector instance uniquely (for example, when monitoring the broker through JMX). If you define more than one `networkConnector` element on a broker, you must set the name in order to ensure that the connector name is unique within the scope of the broker.

`uri`

The discovery agent URI on page 18 that returns which brokers to connect to. In other words, broker A connects to *every* transport URI returned by the discovery agent.

In the preceding example, the static discovery agent URI returns a single transport URI, `tcp://localhost:61002`, which refers to a port opened by one of the transport connectors on broker B.

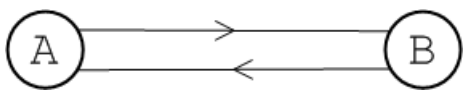
`networkTTL`

The network time-to-live (TTL) attribute specifies the maximum number of hops that a message can make through the broker network. It is almost always necessary to set this attribute, because the default value of 1 would only enable a message to make a single hop to a neighboring broker. For more details, see "Time-to-live" on page 32.

Connectors in each direction

Figure 2.1 on page 26 shows a pair of network connectors in each direction: one from broker A to broker B, and one from broker B to broker A. In this network, there is no restriction on the direction of message flow and messages can propagate freely in either direction. It follows that producers and consumers can arbitrarily connect to either broker in this network.

Figure 2.2. Connectors in Each Direction



In order to create a connector in the reverse direction, from B to A, define a network connector on broker B, as follows:

Example 2.2. Two way connector

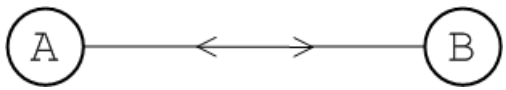
```
<beans ...>
    <broker xmlns="http://activemq.apache.org/schema/core"
            brokerName="brokerB" brokerId="B"... >
        ...
        <networkConnectors>
            <networkConnector name="linkToBrokerA"
                uri="static:(tcp://localhost:61001)"
                networkTTL="3" />
        </networkConnectors>
        ...
        <transportConnectors>
            <transportConnector name="openwire" uri="tcp://0.0.0.0:61002" />
        </transportConnectors>
```

```
    </broker>
</beans>
```

Duplex connector

An easier way to enable message propagation in both directions is by enabling duplex mode on an existing connector. Figure 2.3 on page 28 shows a duplex network connector defined on broker A (where the dot indicates which broker defines the network connector in the figure). The duplex connector allows messages to propagate in both directions, but only one network connector needs to be defined and only *one* network connection is created.

Figure 2.3. Duplex Connector



To enable duplex mode on a network connector, simple set the `duplex` attibute to `true`. For example, to make the network connector on broker A a duplex connector, you can configure it as follows:

Example 2.3. Duplex connector configuration

```
<networkConnectors>
   <networkConnector name="linkToBrokerB"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3"
       duplex="true" />
</networkConnectors>
```

Tip

Duplex mode is particularly useful for cases where a network connection must be established across a firewall, because only one port need be opened on the firewall to enable bi-directional traffic.

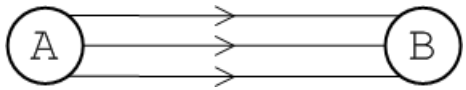
Tip

Duplex mode works particularly well in a hub and spoke network. The spokes only need to know about one hub port and the hub does not need to know any of the spoke addresses (each spoke opens a duplex network connector to the hub).

Multiple connectors

It is also possible to establish multiple connectors between brokers, as long as you observe the rule that each connector has a unique name. Figure 2.4 on page 29 shows an example where three network connectors are established from broker A to broker B.

Figure 2.4. Multiple Connectors



To configure multiple connectors from broker A, use a separate `networkConnector` element for each connector and specify a unique name for each connector, as follows:

```
<networkConnectors>
   <networkConnector name="link01ToBrokerB"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3"
   />
   <networkConnector name="link02ToBrokerB"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3"
   />
   <networkConnector name="link03ToBrokerB"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3"
   />
</networkConnectors>
```

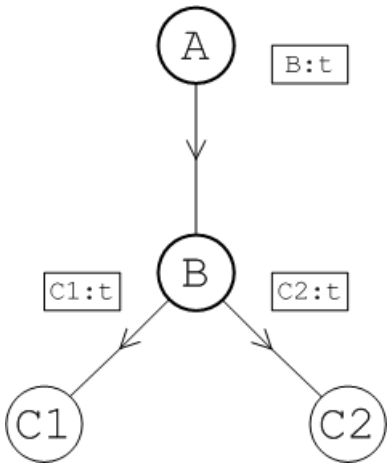
Here are some potential uses for creating multiple connectors between brokers:

- Spreading the load amongst multiple connections.
- Defining separate configuration for topics and queues. That is, you can configure one network connector to transmit queue subscriptions only and another network connector to transmit topic subscriptions only (see "Filtering Messages and Destinations" on page 46).

Conduit subscriptions

By default, after passing through a network connector, subscriptions to the same queue or subscriptions to the same topic are automatically consolidated into a *single* subscription known as a *conduit subscription*. Figure 2.5 on page 30 shows an overview of how the topic subscriptions from two consumers, C1 and C2, are consolidated into a single conduit subscription after propagating from broker B to broker A.

Figure 2.5. Conduit Subscriptions



In this example, each consumer subscribes to the identical topic, `t`, which gives rise to the subscriptions, `C1:t` and `C2:t` in broker B. Both of these subscriptions propagate automatically from broker B to broker A. Because broker A has conduit subscriptions enabled, its network connector consolidates the duplicate subscriptions, `C1:t` and `C2:t`, into a single subscription, `B:t`. Now, if a message on topic `t` is sent to broker A, broker A sends a *single* copy of the message to broker B, to honor the conduit subscription, `B:t`. Broker B then sends a copy of the message to *each* consumer, to honor the topic subscriptions, `C1:t` and `C2:t`.

It is essential to enable conduit subscription in order to avoid duplication of topic messages. Consider what would happen in Figure 2.5 on page 30 if conduit subscription was disabled. In this scenario, two subscriptions, `B:C1:t` and `B:C2:t`, would be registered in broker A. Now, if a message on topic `t` is sent to broker A, broker A would send *two* copies of the message to broker B, to honor the topic subscriptions, `B:C1:t` and `B:C2:t`. Broker B would then send *two* copies of the message to *each* consumer, to honor the topic subscriptions, `C1:t` and `C2:t`. In other words, each consumer would receive the topic message twice.

Conduit subscriptions can optionally be disabled by setting the `conduitSubscriptions` attribute to `false` on the `networkConnector` element. See "Balancing Consumer Load" on page 53 for more details.

Basic Network Concepts

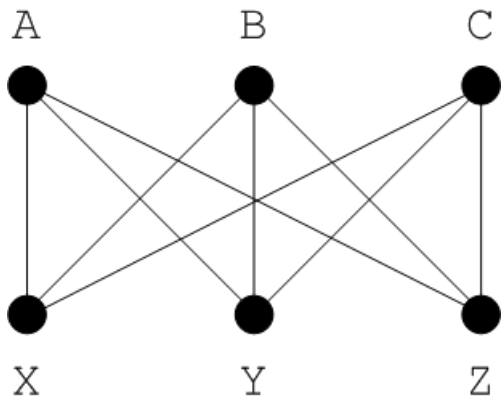
Overview

When discussing networks, it is useful to borrow some basic ideas and definitions from the mathematics of *graph theory*, which is concerned with the description of abstract collections of nodes and connections.

Graph

A *graph* is a mathematical entity consisting of a *vertex set* (analogous to the set of nodes in a network) and an *edge set* (analogous to the set of connections between network nodes). A graph can therefore be used to describe the underlying topology of a network. For example, consider the graph shown in Figure 2.6 on page 31.

Figure 2.6. The Utility Graph



Formally, the utility graph[1] consists of the following:

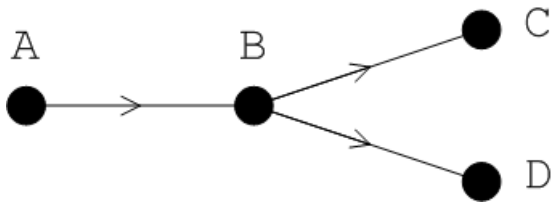
- The *Vertex set*, `{A,B,C,X,Y,Z}`.
- The *Edge set*, `{{A,X},{A,Y},{A,Z},{B,X},{B,Y},{B,Z},{C,X},{C,Y},{C,Z}}`, where each edge is represented by the pair of vertices it joins. For example, the join from vertex `A` to vertex `X` is represented as the edge, `{A,X}`.

[1]The name of this graph derives from a puzzle, where you are asked to find a way to connect each of the three houses, A, B, and C, to the three utilities, X, Y, and Z, without having any of the wires or pipes cross over (in graph theory, a graph that can be drawn without edge crossings is called *planar*).

Digraph

A *directed graph* or *digraph* is a graph that has a direction associated with each edge. In other words, each edge is represented by an ordered pair of vertices, such as `(A,B)`. In a diagram, the order of the vertex pair `(A,B)` is indicated by an arrow pointing from A to B. For example, Figure 2.7 on page 32 shows a simple digraph whose edge set is equal to the set of ordered pairs, `{(A,B),(B,C),(B,D)}`.

Figure 2.7. A Digraph



Given that each simple network connector has a particular direction associated with it (which is the direction of message propagation), it follows that digraphs provide a natural mathematical model for broker network topologies.

Distance between vertices

In graph theory, the *distance*, `d(X,Y)`, between two vertices, `X` and `Y`, is the minimum number of edges that must be traversed in order to get from vertex `X` to vertex `Y`.

For example, if you consider the utility graph shown in Figure 2.6 on page 31, you can see that `d(A,X)=1` and `d(X,Y)=2`. The shortest distance between `X` and `Y` is realised by any of the paths `XAY`, `XBY`, or `XCY`.

The concept of distance is useful in network theory, because it corresponds to the length of the shortest (optimal) route between any two nodes in a network.

Diameter of a graph

The *diameter* of a graph is the greatest distance between any pair of vertices.

For example, the utility graph of Figure 2.6 on page 31 has a diameter of 2.

Time-to-live

The network time-to-live (TTL) is a message property that determines the maximum number of hops that a message can make through a broker network. When a message is originally generated, it is assigned a TTL value (as specified by the `networkTTL` attribute on the corresponding network connector). Each time the

message traverses a hop from one broker to the next, the message's TTL property is decremented by 1 and when the TTL property reaches 0, the message cannot be forwarded to another broker (though it *can* be sent to a local consumer).

It follows from this definition that, if the graph-theoretical distance between two brokers is greater than a message's TTL value, it will not be possible for the message to travel all the way from one broker to the other. In fact, it is possible to state the following general rule for the network TTL: *to ensure that a message can reach* any *node in a broker network, the message's TTL value must be greater than or equal to the diameter of the network*.

Graph topologies

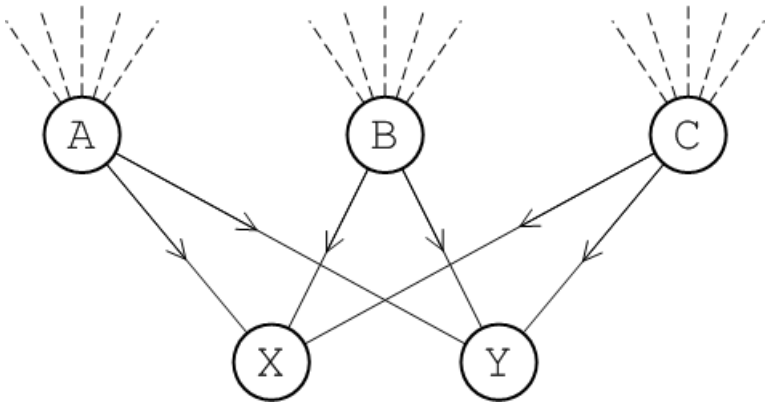
The following examples illustrate some of the common graph topologies encountered real-world networks:

- "Concentrator topology" on page 33.
- "Hub and spokes topology" on page 34.
- "Tree topology" on page 35.
- "Mesh topology" on page 36.
- "Complete graph" on page 37.

Concentrator topology

If you anticipate that your system will have a large number of incoming connections that would overwhelm a single broker, you can deploy a concentrator topology to deal with this scenario, as shown in Figure 2.8 on page 34.

Figure 2.8. Concentrator Topology

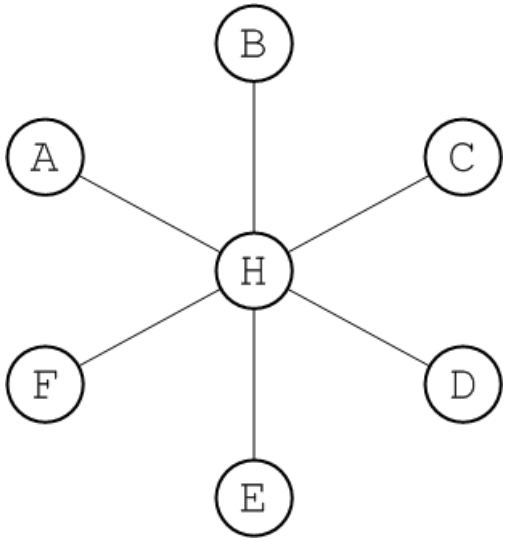


The idea of the concentrator topology is that you deploy brokers in two (or more) layers in order to funnel incoming connections into a smaller collection of services. The first layer consists of a relatively large number of brokers, with each broker servicing a large number of incoming connections (from producers `P1` to `Pn`). The next layer consists of a smaller number of brokers, where each broker in the first layer connects to all of the brokers in the second layer. With this topology, each broker in the second layer can receive messages from *any* of the producers.

Hub and spokes topology

The hub and spokes, as shown in Figure 2.9 on page 35, is a topology that is relatively easy to set up and maintain. The edges in this graph are all assumed to represent duplex network connectors.

Figure 2.9. Hub and Spoke Topology

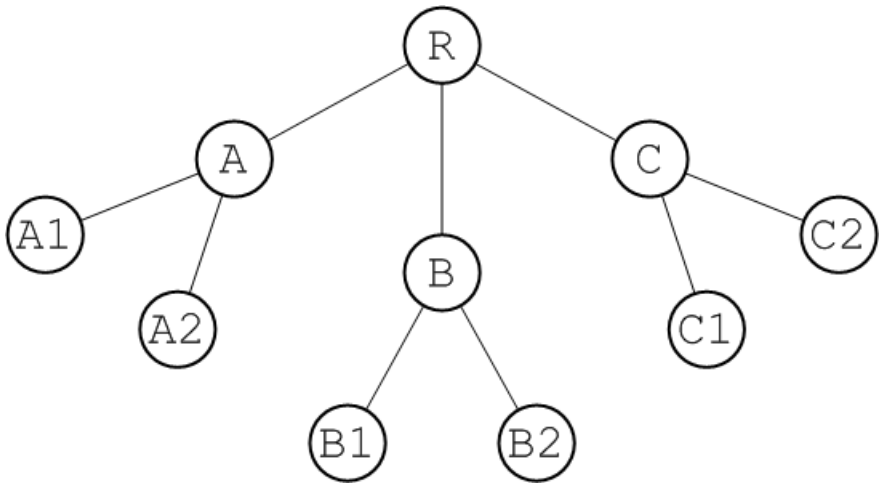


This topology is relatively robust. The only critical element is the hub node, so you would need to focus your maintenance efforts on keeping the hub up and running. Routes are determinate and the diameter of the network is always 2, no matter how many nodes are added.

Tree topology

The tree, as shown in Figure 2.10 on page 36, is a topology that arises naturally when a physical network grows in an informal manner.

Figure 2.10. Tree Topology

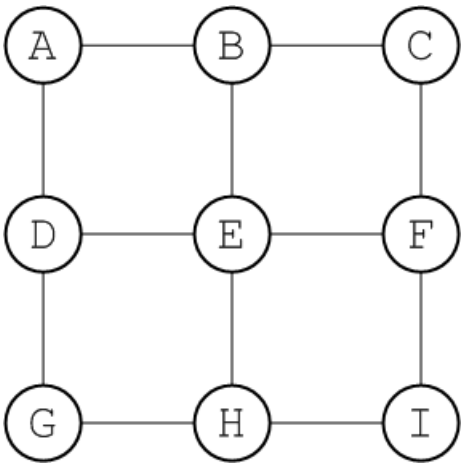


For example, if the network under consideration is an ethernet LAN, `R` could represent the hub in the basement of the IT department's building and `A` could represent a router in the ground floor of another building. If you want to extend the LAN to the first and second floor of building `A`, you are unlikely to run dedicated cables back to the IT hub for each of these floors. It is more likely that you will simply plug a second tier of routers, `A1` and `A2`, into the existing router, `A`, on the ground floor. In this way, you effectively add another layer to the tree topology.

Mesh topology

The mesh, as shown in Figure 2.11 on page 37, is a topology that arises naturally in a geographic network, when you decide to link together neighbouring hubs.

Figure 2.11. Mesh Topology

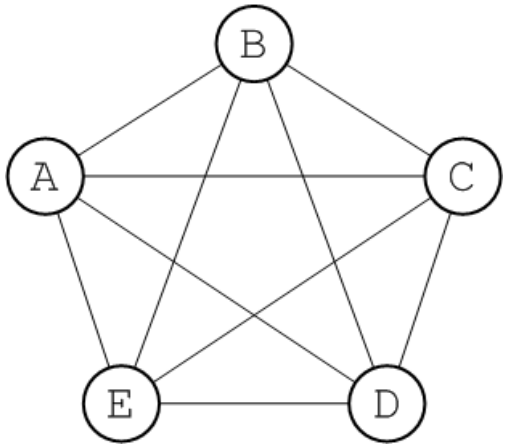


The diameter of a mesh increases whenever you add a node to its periphery. You must, therefore, be careful to set the network TTL sufficiently high that your network can cope with expansion. Alternatively, you could set up some mechanism for the central management of broker configurations. This would enable you to increase the network TTL for all of the brokers simultaneously.

Complete graph

In graph theory, the *complete graph on n vertices* is the graph with n vertices that has edges joining every pair of vertices. This graph is denoted by the symbol, K_n. For example, Figure 2.12 on page 38 shows the graph, K_5.

Figure 2.12. The Complete Graph, K_5



Every complete graph has a diameter of 1. Potentially, a network that is a complete graph could be difficult to manage, because there are many connections between broker nodes. In practice, though, it is relatively easy to set up a broker network as a complete graph, if you define all of the network connectors to use a multicast discovery agent (see "Multicast discovery agent" on page 20).

Dynamic and Static Propagation

Overview

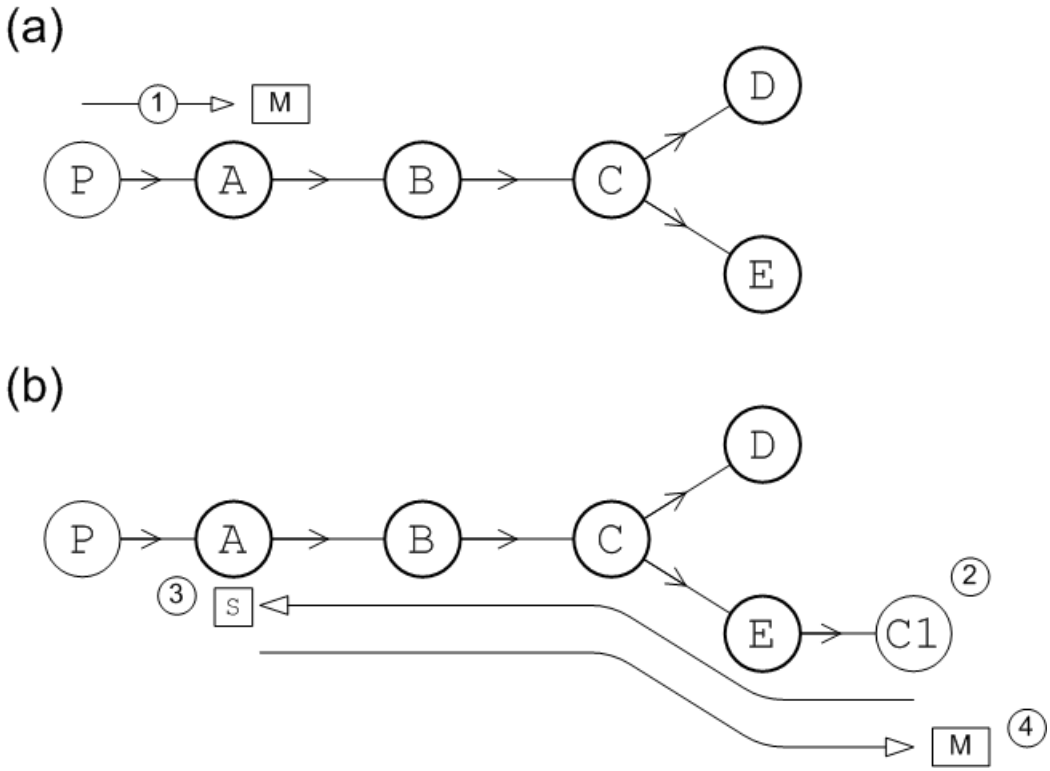
The fundamental purpose of a broker network is to route messages to their intended recipients, which are consumers that could be attached at any point in the network. The peculiar difficulty in devising routing rules for a messaging network is that messages are sent to an *abstract* destination rather than a *physical* destination. In other words, a message might be sent to a specific queue, but that gives you no clue as to which broker or which consumer that message should ultimately be sent to. Contrast this with the Internet Protocol (IP), where each message packet includes a header with an IP address that references the physical location of the destination host.

Because of the special nature of routing in a messaging system, the propagation of messages must be inherently dynamic. That is, the broker network must keep track of the active consumers attached to the network and the routing of messages is governed by the real-time transmission of advisory messages (subscriptions).

Dynamic propagation

Figure 2.13 on page 40 illustrates how dynamic propagation works for messages sent to a queue. The broker connectors in this network are simple (non-duplex).

Figure 2.13. Dynamic Propagation of Queue Messages



The dynamic message propagation in this example proceeds as follows:

1. As shown in part (a), initially, there are *no* consumers attached to the network. A producer, `P`, connects to broker A and starts sending messages to a particular queue, `TEST.FOO`. Because there are no consumers attached to the network, all of the messages accumulate in broker A. The messages do *not* propagate any further at this time.

2. As shown in part (b), a consumer, C, now connects to the network at broker E and subscribes to the same queue, `TEST.FOO`, to which the producer is sending messages.

3. The consumer's subscription, `S`, propagates through the broker network, following the reverse arrow direction, until it reaches broker A.

4. After broker A receives the subscription, `s`, it knows that it can send the messages accumulated in the queue, `TEST.FOO`, to the consumer, C. Based on the information in the subscription, `s`, broker A sends messages along the path ABCE to reach consumer C.

Static propagation

Static propagation refers to message propagation that occurs in the *absence* of subscription information. Sometimes, because of the way a broker network is set up, it can make sense to move messages between brokers, even when there is no relevant subscription information.

Static propagation is configured by specifying the queue (or queues) that you want to statically propagate. Into the relevant `networkConnector` element, insert `staticallyIncludedDestinations` as a child element and then list the queues and topics you want to propagate using the `queue` and `topic` child elements. For example, to specify that messages in the queue, `TEST.FOO`, are statically propagated from A to B, you would define the network connector in broker A's configuration as follows:

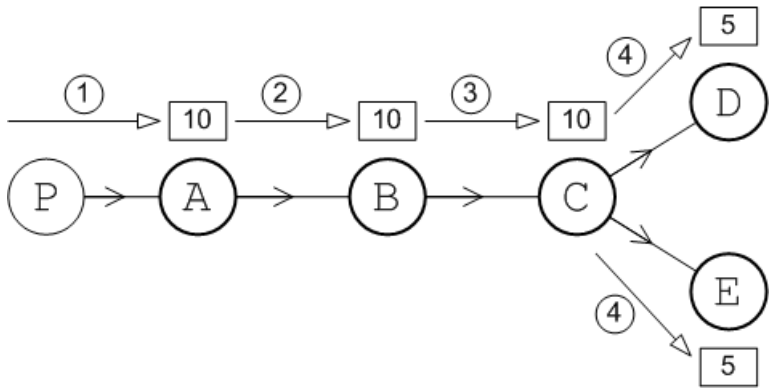
```
<networkConnectors>
    <networkConnector name="linkToBrokerB"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3">
        <staticallyIncludedDestinations>
            <queue physicalName="TEST.FOO"/>
        </staticallyIncludedDestinations>
    </networkConnector>
</networkConnectors>
```

Note

You cannot use wildcards when specifying statically included queue names or topic names.

Consider the network shown in Figure 2.14 on page 42. This network is set up so that consumers only attach to broker D or to broker E Messages sent to the queue, `TEST.FOO`, are configured to propagate statically on all on all of the network connectors, `(A,B)`, `(B,C)`, `(C,D)`, and `(C,E)`.

Figure 2.14. Static Propagation of Queue Messages



The static message propagation in this example proceeds as follows:

1. Initially, there are *no* consumers attached to the network. A producer, `P`, connects to broker A and sends 10 messages to the queue, `TEST.FOO`.

2. Because the network connector, `(A,B)`, has enabled static propagation for the queue, `TEST.FOO`, the 10 messages on broker A are forwarded to broker B.

3. Likewise, because the network connector, `(B,C)`, has enabled static propagation for the queue, `TEST.FOO`, the 10 messages on broker B are forwarded to broker C.

4. Finally, because the network connectors, `(C,D)` and `(C,E)`, have enabled static propagation for the queue, `TEST.FOO`, the 10 messages on broker C are alternately sent to broker D and broker E. In other words, the brokers, D and E, receive every second message. Hence, at the end of the static propagation, there are 5 messages on broker D and 5 messages on broker E.

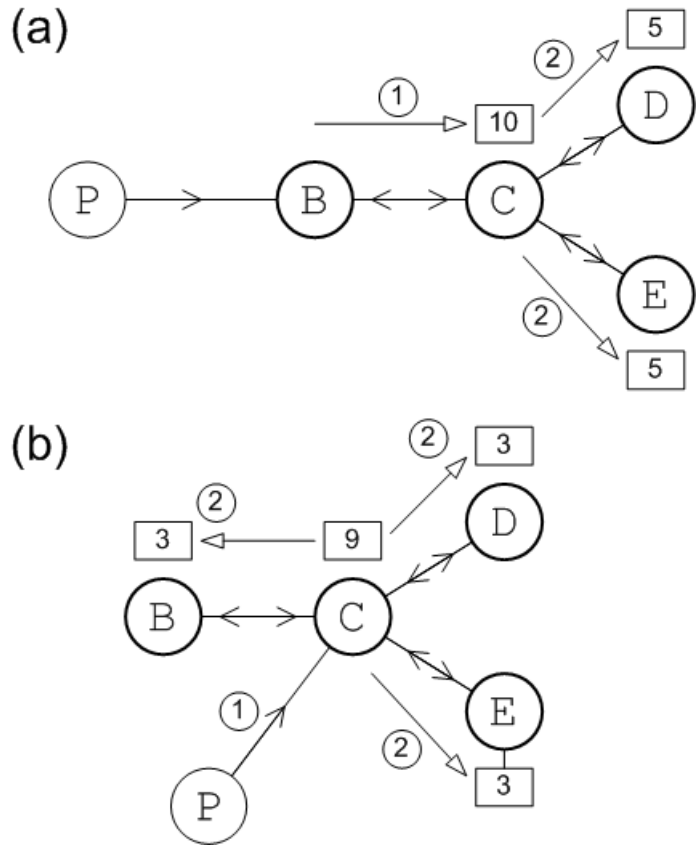
Note

Using the preceding static configuration, it is possible for messages to get stuck in a particular broker. For example, if a consumer now connects to broker E, it will receive the 5 messages stored on broker E, but it will *not* receive the 5 messages stored on broker D. The messages remain stuck on broker D until a consumer connects directly to it.

Duplex mode and static propagation

It is also possible to use static propagation in combination with duplex connectors. In this case, messages can propagate statically in *either* direction through the duplex connector. For example, Figure 2.15 on page 43 shows a network of four brokers, B, C, D, and E, linked by duplex connectors. All of the connectors have enabled static propagation for the queue, `TEST.FOO`.

Figure 2.15. Duplex Mode and Static Propagation



In part (a), the producer, P, connects to broker B and sends 10 messages to the queue, `TEST.FOO`. The static message propagation then proceeds as follows:

1. Because the duplex connector, `{B,C}`, has enabled static propagation for the queue, `TEST.FOO`, the 10 messages on broker B are forwarded to broker C.

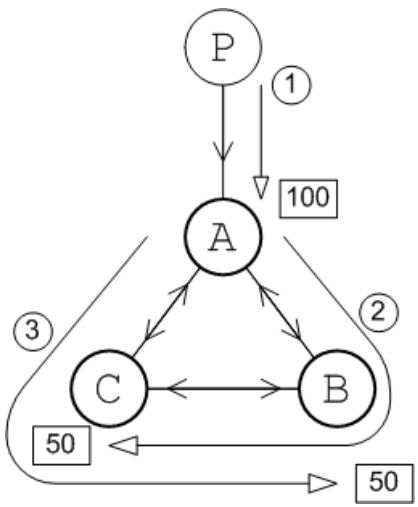
2. Because the duplex connectors, `{C,D}` and `{C,E}`, have enabled static propagation for the queue, `TEST.FOO`, the 10 messages on broker C are alternately sent to broker D and broker E. At the end of the static propagation, there are 5 messages on broker D and 5 messages on broker E.

In part (b), the producer, P, connects to broker C and sends 9 messages to the queue, `TEST.FOO`. Because static propagation is enabled on all of the connectors, broker C sends messages alternately to B, D, and E. At the end of the static propagation, there are 3 messages on broker B, 3 messages on broker D, and 3 messages on broker E.

Self-avoiding paths

Brokers implement a strategy of *self-avoiding paths* in order to prevent pathalogical routes from occurring in a statically configured broker network. For example, consider what could happen, if a closed loop occurs in a network with statically configured duplex connectors. If the brokers followed a strategy of simply forwarding messages to a neighbouring broker (or brokers), messages could end up circulating around the closed loop for ever. This does *not* happen, however, because the broker network applies a strategy of self-avoiding paths to static propagation. For example, Figure 2.16 on page 44 shows a network consisting of three brokers, A, B, and C, linked by statically configured duplex connectors. The path ABCA forms a closed loop in this network.

Figure 2.16. Self-Avoiding Paths



The static message propagation in this example proceeds as follows:

1. The producer, P, connects to broker A and sends 100 messages to the queue, `TEST.FOO`.
2. The 100 messages on broker A are alternately sent to broker B and broker C. The 50 messages sent to broker B are immediately forwarded to broker C, but at this point the messages stop moving and remain on broker C. The self-avoiding path strategy dictates that messages can *not* return to a broker they have already visited.
3. Similarly, the 50 messages sent from broker A to broker C are immediately forwarded to broker B, but do not travel any further than that.

brokerId and self-avoiding paths

Fuse Message Broker uses broker ID values (set by the `broker` element's `brokerId` attribute) to figure out self-avoiding paths. By default, the broker ID value is generated dynamically and assigned a new value each time a broker starts up. If your network topology relies on self-avoiding paths, however, this default behavior is *not* appropriate. If a broker is stopped and restarted, it would rejoin the network with a different broker ID, which confuses the self-avoiding path algorithm and can lead to stuck messages.

In the context of a broker network, therefore, it is recommended that you set the broker ID explicitly on the `broker` element, as shown in the following example:

```
<broker xmlns="http://activemq.apache.org/schema/core"
        brokerName="brokerA" brokerId="A"... >
    ...
</broker>
```

Note

Make sure you always specify a broker ID that is unique within the current broker network.

Filtering Messages and Destinations

Overview

Typically, one of the basic tasks of managing a broker network is to partition the network so that certain queues and topics are restricted to a sub-domain, while messages on other queues and topics are allowed to cross domains. This kind of domain management can be achieved by applying filters at certain points in the network. Fuse Message Broker lets you define filters on network connectors and message selectors on consumers in order to control the flow of messages throughout the network.

Destination wildcards

The following characters can be used to define wildcard matches for topic names and queue names:

`.`	Separates segments in a path name.
`*`	Matches any single segment in a path name.
`>`	Matches any number of segments in a path name.

Wildcards are meant to be used with destination names that have a segmented structure, like a path name—for example, `PRICE.STOCK.NASDAQ.IBM` or `PRICE.STOCK.NYSE.SUNW`—where the segments are delimited by the `.` character. The following table shows some examples of destination wildcards and describes what names they would match.

Destination wildcard	**What it matches**
`PRICE.>`	Any price for any product on any exchange.
`PRICE.STOCK.>`	Any price for a stock on any exchange.
`PRICE.STOCK.NASDAQ.*`	Any stock price on NASDAQ.
`PRICE.STOCK.*.IBM`	Any IBM stock price on any exchange.

Filtering dynamic destinations

It is possible to filter the messages that pass through a network connector by specifying destinations to include and destinations to exclude. The following child elements of `networkConnector` are used to filter dynamic destinations:

`dynamicallyIncludedDestinations`
: Explicitly specifies a list of included destinations, where the included destinations are specified using `queue` child elements and `topic` child elements (wildcards are allowed). Destinations matching this list and *only these destinations* are propagated dynamically by the network connector (provided they are not

blocked by the list specified in `excludedDestinations`). An empty list allows *all* destinations to be propagated dynamically.

`excludedDestinations`
: Specifies a list of excluded destinations, where the excluded destinations are specified using `queue` child elements and `topic` child elements (wildcards are allowed). Destinations matching this list are always blocked by the network connector.

Example 2.4 on page 47 shows a network connector that is configured to filter dynamic destinations. In this example, the connector transmits stock prices from any exchange except the NYSE and transmits orders to trade stocks for any exchange except the NYSE.

Example 2.4. Connector with Filtered Destinations

```
<networkConnectors>
    <networkConnector name="linkToBrokerB"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3">
        <dynamicallyIncludedDestinations>
            <queue physicalName="TRADE.STOCK.>"/>
            <topic physicalName="PRICE.STOCK.>"/>
        </dynamicallyIncludedDestinations>
        <excludedDestinations>
            <queue physicalName="TRADE.STOCK.NYSE.*"/>
            <topic physicalName="PRICE.STOCK.NYSE.*"/>
        </excludedDestinations>
    </networkConnector>
</networkConnectors>
```

Filtering and static destinations

There is no need to filter statically propagated destinations, because destinations are only transmitted statically when they are listed explicitly in a `staticallyIncludedDestinations` element.

Message selectors

Fuse Message Broker supports standard JMS selectors, which enable consumers to filter messages from a particular destination by testing the contents of a message's JMS headers. When a consumer subscribes to a particular destination, the standard JMS API can be used to specify a selector (see javax.jms.Message[2] for more details).

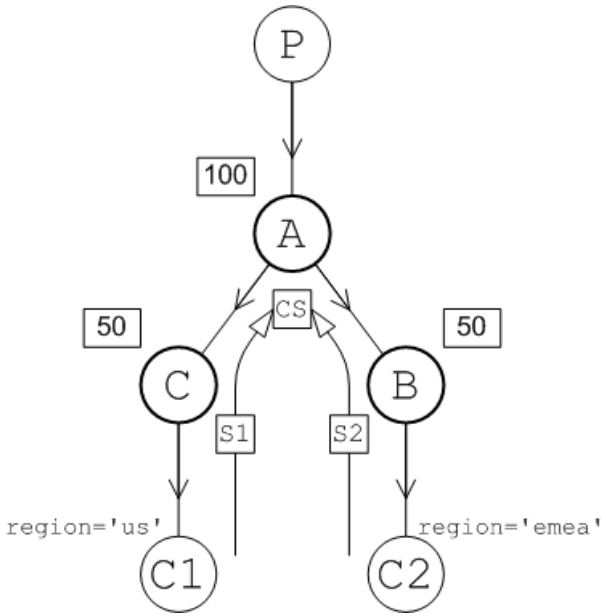
If your consumers use JMS selectors, you should be aware that some interaction can occur between JMS selectors and conduit subscriptions. In general, JMS selectors are always applied by the consumer's local

[2] http://java.sun.com/j2ee/1.4/docs/api/javax/jms/Message.html

broker (that is, the broker to which the consumer is directly connected). But if the conduit subscriptions option is enabled (which it is by default), JMS selector information is omitted from any conduit subscriptions transmitted to a remote broker. This implies that the remote broker (or brokers) do not perform any filtering based on JMS selectors. There are certain scenarios where this behavior can cause problems.

For example, consider the broker network shown in Figure 2.17 on page 48, where conduit subscription is enabled throughout the network. Both of the consumers, C1 and C2, subscribe to the same queue and they also define JMS selectors as follows: C1 selects messages for which the `region` header is equal to `us`; and C1 selects messages for which the `region` header is equal to `emea`.

Figure 2.17. Interaction between JMS Selector and Conduit Subscription



Now consider what happens when a producer, P, connects to broker A and starts sending messages to the queue. The consumer subscriptions, `s1` and `s2`, automatically propagate to broker A. But because these subscriptions are both on the same queue and the conduit subscriptions option is enabled, broker A combines the subscriptions into a single conduit subscription, `cs`, which does *not* include any selector details. When P starts sending messages to the queue, broker A forwards the messages alternately to broker B and broker C, *without* checking whether the messages satisfy the relevant selectors. If you are particularly unlucky, you might find that all of the messages for region `emea` end up on broker B and all of the messages for region `us` end up on broker C. In this case, none of the messages could be consumed.

Optimizing Routes

Overview

If you are using a network topology such as a hub-and-spoke (see Figure 2.9 on page 35) or a tree (see Figure 2.10 on page 36), the network route is inherently deterministic and you do not need to concern yourself with choosing an optimum route. For other topologies, however, it is possible to have multiple alternative routes joining a producer to a consumer across the network. In such cases, it is usually preferable to simplify the routing behavior, so that an optimum route is preferred by the network.

Choosing the closest consumer

Certains kinds of network—for example, "Hub and spokes topology" on page 34 and "Tree topology" on page 35—have *determinate* routes. That is, there exists a unique route between any two brokers in the network.

If your network is a mesh, on the other hand, you might find that there are multiple routes joining some pairs of brokers. For such indeterminate networks, it is normally preferable for messages to propagate along the *shortest* route, in order to maximize the efficiency of the network. To ensure that the shortest route is preferred, enable the `decreaseNetworkConsumerPriority` option on all of the connectors in the network (the default is `false`). For example, you can enable this option on a network connector as follows:

```
<networkConnectors>
    <networkConnector name="linkToBrokerB"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3"
       decreaseNetworkConsumerPriority="true" />
</networkConnectors>
```

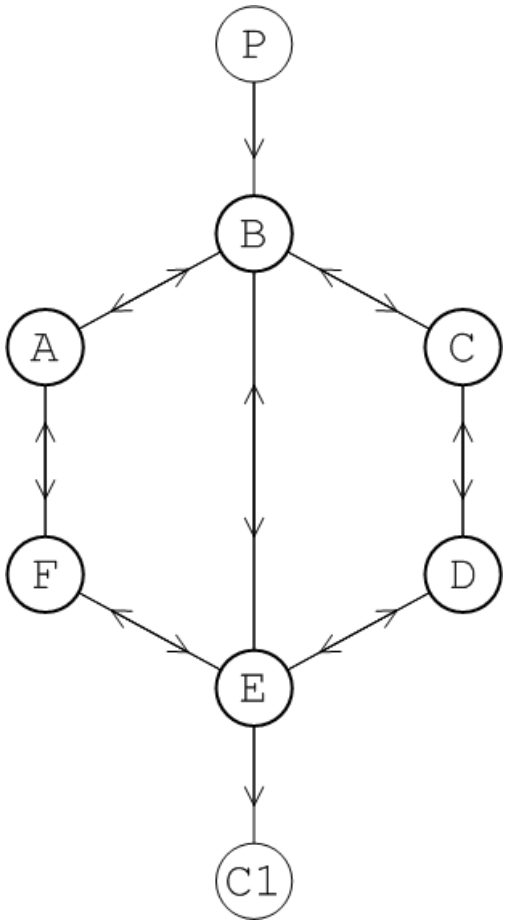
When `decreaseNetworkConsumerPriority` is set to `true`, the priority is set as follows:

- Local consumers (attached directly to the broker) have a priority of `0`.
- Network subscriptions have an initial priority of `-5`.
- The priority of a network subscription is reduced by `1` for every network hop that it traverses

A broker sends messages preferentially to the subscription with the highest priority, but if the prefetch buffer is full, the broker will divert messages to the subscription with the next highest priority. If multiple subscriptions have the same priority, the broker distributes messages equally between those subscriptions.

Figure 2.18 on page 50 illustrates the effect of setting this option in a broker network.

Figure 2.18. Shortest Route in a Mesh Network



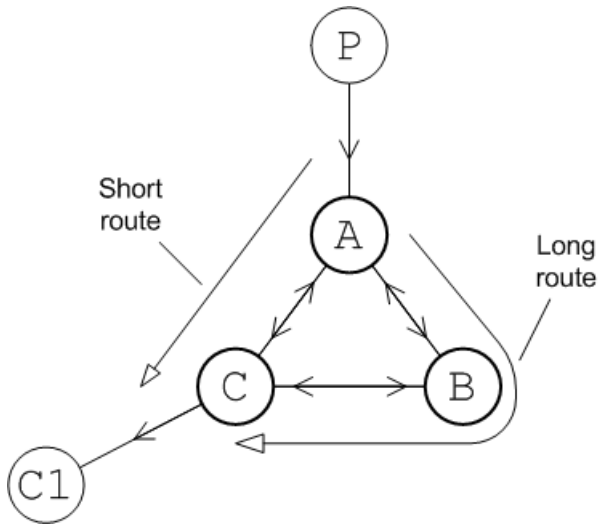
In this network, there are three alternative routes connecting producer P to consumer C1—these are `PBAFEC1` (three broker hops), `PBEC1` (one broker hop), and `PBCDEC1` (three broker hops). When `decreaseNetworkConsumerPriority` is enabled, the route, `PBEC1`, has highest priority, so messages from P to C1 are sent preferentially along this route.

Eliminating duplicate routes

In some cases, setting `decreaseNetworkConsumerPriority` to `true` is not enough to ensure deterministic routing in a network. Consider the network of brokers, A, B, and C, shown in Figure 2.19 on page 51. In this

scenario, a producer, P, (which writes messages to queue, `foo`) connects to broker A and a consumer, C1, (which reads messages from queue, `foo`) connects to broker B. The network TTL is equal to `2`, so two alternative routes are possible: the short route, `PABC1`, and the long route, `PACBC1`.

Figure 2.19. Duplicate Subscriptions in a Network



Now, if you set `decreaseNetworkConsumerPriority` to `true`, the short route is preferred. So, messages are propagated along the route `PABC1`. However, under heavy load conditions, the short route, `PABC1`, can become overloaded and in this case the broker, A, will fall back to the long route, `PACBC1`. The problem with this scenario is that when the consumer, C1, shuts down, it can lead to messages getting stuck on broker C. In order to avoid this problem, it is recommended that you set the `suppressDuplicateQueueSubscriptions` option to `true` on all of the network connectors in you network. For example, you can set this option as follows:

```
<networkConnectors>
    <networkConnector name="linkToBrokerB"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3"
       decreaseNetworkConsumerPriority="true"
       suppressDuplicateQueueSubscriptions="true"/>
</networkConnectors>
```

The effect of enabling this option is that the broker allows only a *single* queue subscription to be created for a given consumer. This means that only a single route can be created between a producer and a consumer, so that the routing becomes fully deterministic. In general, it is recommended that you enable both the `decreaseNetworkConsumerPriority` option and the `suppressDuplicateQueueSubscriptions` option together.

Note

In the example shown in Figure 2.19 on page 51, you could have suppressed the long route by reducing the network TTL to 1. Normally, however, in a large network you do not have the option of reducing the network TTL arbitrarily. The network TTL has to be large enough for messages to reach the most distant brokers in the network.

brokerId and duplicate routes

Fuse Message Broker uses broker ID values (set by the `broker` element's `brokerId` attribute) to figure out duplicate routes. In order for the elimination of duplicate routes to work reliably, it is recommended that you set the broker ID explicitly on the `broker` element for each broker in the network, as shown in the following example:

```
<broker xmlns="http://activemq.apache.org/schema/core"
        brokerName="brokerA" brokerId="A"... >
    ...
</broker>
```

Note

Make sure you always specify a broker ID that is unique within the current broker network.

Balancing Consumer Load

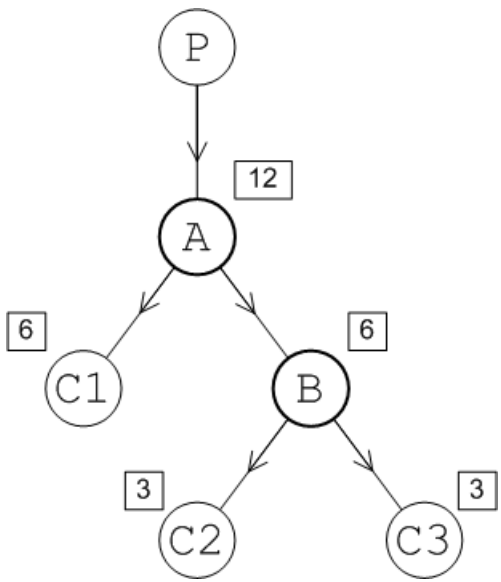
Overview

Depending on the topology of your broker network, the default conduit subscription behavior can sometimes be detrimental to load balancing on the consumer side. This section describes the nature of the problem and explains how to alter the network configuration to achieve optimum load balancing.

Default load behavior

When the conduit subscriptions option is enabled (which it is by default), this can result in an uneven distribution of messages between consumers that subscribe to a particular queue. Figure 2.20 on page 53 illustrates how this uneven distribution can come about.

Figure 2.20. Message Flow when Conduit Subscriptions Enabled



Assume that the consumers, C1, C2, and C3, all subscribe to the same queue, `TEST.FOO`. Now consider what happens when producer, P, connects to broker A and sends 12 messages to the queue, `TEST.FOO`. Because conduit subscriptions is enabled, broker A sees only a single (conduit) subscription from broker B. Broker A also sees a single subscription from consumer C1. So, broker A will send messages alternately to C1 and B, sending a total of 6 messages to C1 and 6 messages to B. Now broker B sees two subscriptions, from C2 and

C3 respectively. So, broker B will send messages alternately to C2 and C3, sending a total of 3 message to C2 and 3 messages to C3.

In the end, the distribution of messages amongst the consumers is 6, 3, 3, which is not optimally load balanced.

Disabling conduit subscriptions

If you want to improve the load balancing behavior for queues, you can disable conduit subscriptions by setting `conduitSubscriptions` to `false`. For example, you can disable conduit subscriptions on a broker connector as follows:

```
<networkConnectors>
    <networkConnector name="linkToBrokerB"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3"
       conduitSubscriptions="false" />
</networkConnectors>
```

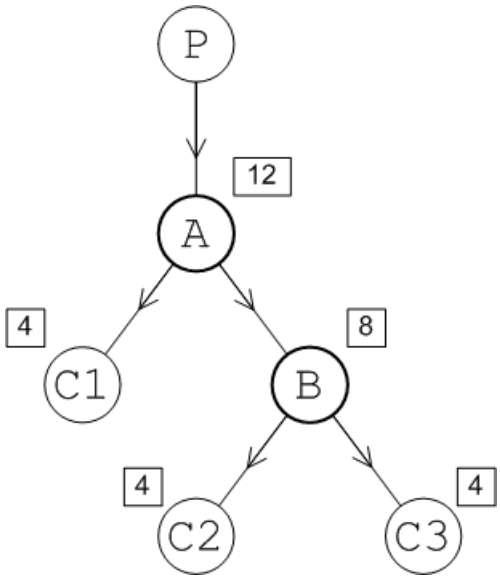
Warning

Be careful, if you are using topics. Disabling conduit subscriptions can lead to duplicate topic messages—see "Conduit subscriptions" on page 29.

Balanced load behavior

When the conduit subscriptions option is disabled, you can achieve optimal distribution of messages between consumers that subscribe to a particular queue. Figure 2.21 on page 55 illustrates the message flow in this case.

Figure 2.21. Message Flow when Conduit Subscriptions Disabled



Assume that the consumers, C1, C2, and C3, all subscribe to the same queue, `TEST.FOO`. Now consider what happens when producer, P, connects to broker A and sends 12 messages to the queue, `TEST.FOO`. Because conduit subscriptions is disabled, broker A sees *two* subscriptions on broker B and a single subscription from consumer C1. So, by alternating between all the subscriptions, broker A ends up sending 4 messages to C1 and 8 messages to broker B. Broker B then sends 4 messages each to consumers C2 and C3.

In the end, the distribution of messages amongst the consumers is 4, 4, 4, which is optimally balanced.

Separate connectors for topics and queues

In some cases, you might need to *disable* conduit subscriptions for queues (in order to optimize load balancing), but also *enable* conduit subscriptions for topics (to avoid duplicate topic messages). You cannot configure this using a single network connector, because the `conduitSubscriptions` flag applies simultaneously to queues and topics. On the other hand, it is possible to configure topics and queues differently, if you create multiple network connectors: one for queues and another for topics (see Figure 2.4 on page 29).

Example 2.5 on page 56 shows how to configure separate network connectors for topics and queues. The `queuesOnly` network connector, which has conduit subscriptions enabled, is equipped with a filter that transmits only queue messages. The `topicsOnly` network connector, which has conduit subscriptions disabled, is equipped with a filter that transmits only topic messages.

Example 2.5. Separate Configuration of Topics and Queues

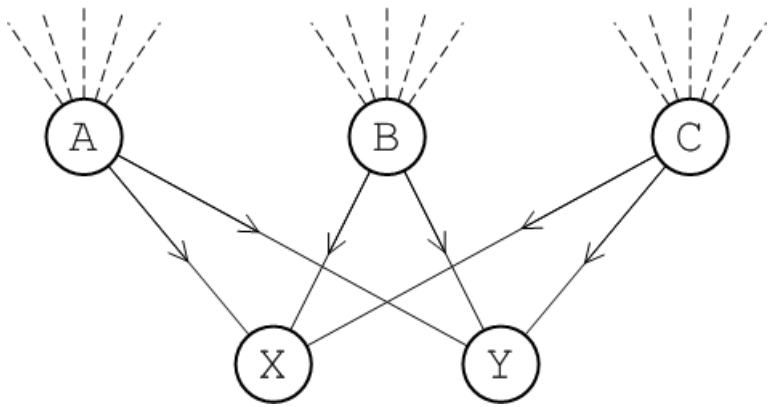
```
<networkConnectors>
    <networkConnector name="queuesOnly"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3"
       conduitSubscriptions="false" >
        <dynamicallyIncludedDestinations>
            <queue physicalName=">"/>
        </dynamicallyIncludedDestinations>
    </networkConnector>

    <networkConnector name="topicsOnly"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3"
       conduitSubscriptions="true" >
        <dynamicallyIncludedDestinations>
            <topic physicalName=">"/>
        </dynamicallyIncludedDestinations>
    </networkConnector>
</networkConnectors>
```

Managing Producer Load

Concentrator topology

For greater scalability on the producer side, you might want to spread the message load across multiple brokers. This can be achieved by setting up a broker network, as described in "Broker Networks" on page 23. For the purpose of spreading the load across brokers, one of the most useful topologies is the concentrator topology, as shown in Figure 2.22 on page 57.

Figure 2.22. Load Balancing with the Concentrator Topology



The concentrator topology consists of two layers of brokers, as follows:

- The first layer of brokers, A, B, and C, accepts connections from message producers and specializes in receiving incoming messages;
- The second layer of brokers, X and Y, accepts connections from message consumers and specializes in sending messages to the consumers.

With this topology, the first layer of brokers, A, B, and C, can focus on managing a large number of incoming producer connections. The received messages are consolidated within the brokers before being passed through a relatively small number of network connectors to the second layer, X and Y. The brokers, X and Y, only need to deal with a relatively small number of connections (assuming the number of consumers is small). If the number of consumers is large, you could add a third layer of brokers to fan out and handle the consumer connections.

Client configuration

When connecting to a broker network, producers and consumers must be distributed across the available brokers. For example, in the case of a producer connecting to the concentrator topology shown in Figure 2.22 on page 57, the producer should connect using a failover URL that shares the load across the brokers in the first layer, A, B, and C. Assuming that the brokers are running on separate hosts—`brokerA`, `brokerB`, and `brokerC`—and assuming that they all listen on IP port 61616, the producers should use the following failover URL to connect to the broker network:

```
failover://(tcp://brokerA:61616,tcp://brokerB:61616,tcp://brokerC:61616)
```

By default, a producer will randomly select one of the URLs in the failover list and attempt to connect to it. If the first connection attempt fails, the producer will try the other URLs in turn. For more details of the failover protocol, see "Failover Protocol" on page 12.

Chapter 3. Load Balancing

This chapter describes how broker networks can be applied to the problem of load balancing in a messaging system.

Load Balancing over Consumers

Queue consumers

Multiple consumers attached to a JMS queue automatically obey *competing consumer* semantics. That is, each message transmitted by the queue is consumed by *one consumer only*. Hence, if you want to scale up load balancing on the consumer side, all that you need to do is attach extra consumers to the queue. The competing consumer semantics of the JMS queue then automatically ensures that the queue's messages are evenly distributed amongst the attached consumers.

Chapter 4. Fault Tolerance

Fault tolerance in the context of Fuse Message Broker means that, in case of a broker failure, there is a broker on standby that is ready to take over from the failed broker, with no loss of data. There are thus two fundamental requirements that the fault tolerant messaging system must satisfy: minimum downtime; and avoidance of data loss when a broker fails. Fuse Message Broker addresses these fault tolerant requirements using a master/slave cluster.

Master/Slave Patterns

Introduction to Master/Slave Clusters

Overview

A master/slave failover pattern consists of a cluster of brokers where one broker (the *master*) is currently active and one or more brokers (the *slaves*) are on hot standby, ready to take over whenever the master fails or shuts down. Note that a master/slave cluster is *not* the same thing as a network of brokers. The brokers in a master/slave cluster are never linked using a network connector.

Comparison of master/slave patterns

Table 4.1 on page 63 provides an overview and comparison of the different master/slave patterns supported in Fuse Message Broker.

Table 4.1. Master/Slave Patterns

Pattern	Requirements	Advantages	Disadvantages
Pure master/slave	*None*	No central point of failure.	Requires manual restart. Only one slave supported.
Shared file system master/slave	A shared file system.	Unlimited number of slaves. Automatic recovery of old masters.	*None*
JDBC master/slave	A shared database.	Unlimited number of slaves. Automatic recovery of old masters.	Cannot use high performance journal.

Configuring clients of the master/slave pair

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 this logic to clients is easy. You set them up to connect to the cluster using the failover protocol which has built in reconnect logic. For example, assuming that 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 failover URL shown in Example 4.1 on page 63 for your clients.

Example 4.1. Failover URL for Connecting to a Master/Slave Cluster

```
failover://(tcp://masterhost:61616,tcp://slavehost:61616)?randomize=false
```

Setting the `randomize` option to `false` ensures that the failover URL tries to connect to the master before the slave. This can speed up the initial connections in pure master/slave set ups.

For more information on using the failover protocol see "Failover Protocol" in *Broker Client Connectivity Guide*.

Pure Master/Slave

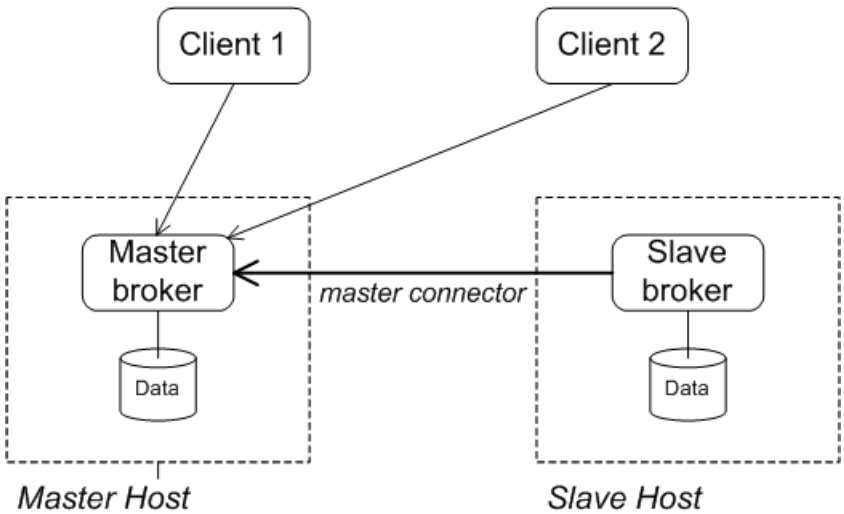
Overview

The pure master/slave failover pattern replicates data between a pair of brokers using a dedicated connection. The advantage of this approach is that it works independently of the persistence layer: it does not require a shared database or a shared file system.

Initial state of the master/slave topology

Figure 4.1 on page 65 shows the initial state of the pure master/slave topology.

Figure 4.1. Pure Master/Slave Initial State



In this topology, the *master broker* is an ordinary broker with no special configuration. Typicaly, the master is configured with some form of persistent storage, but this is *not* a requirement.

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

Producer and consumer clients are configured with a failover URL that tries to connect first of all to the master broker (see "Configuring clients of the master/slave pair" on page 63).

The master broker will respond to a client only after a message exchange has been successfully passed to the slave. For example, a commit in a client transaction will not complete until both the master and the slave have processed the commit.

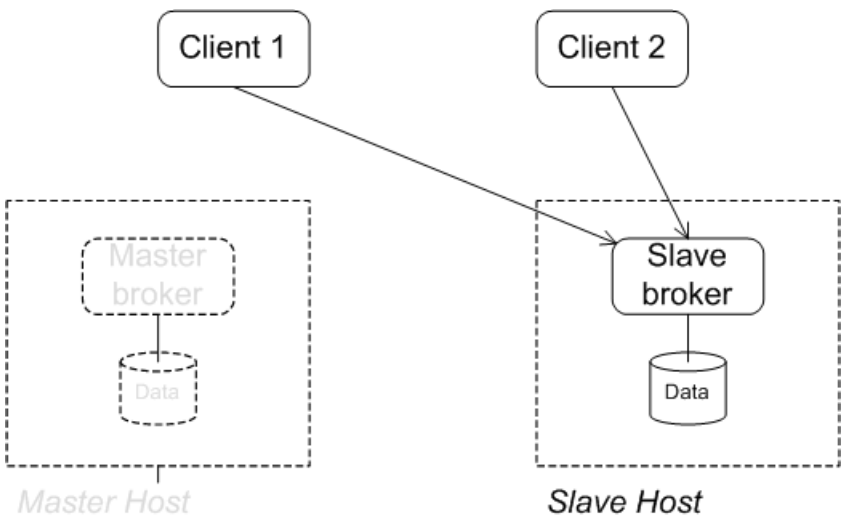
After failure of the master

In the event of the master failing, the slave supports two alternative modes of operation, as follows:

- *Slave broker takes over*—in this case, the slave starts up all of its transport connectors and network connectors and takes the place of the master broker. Clients must be configured to fail over to the slave.
- *Slave broker closes down*—in this case, there is no longer any broker available. The sole purpose of the slave is to preserve a copy of the messaging data, in case there is a catastrophic hardware failure on the master host.

Figure 4.2 on page 66 shows the state of the system after the master broker has failed, assuming that the slave broker takes over from the master.

Figure 4.2. Pure Master/Slave after Master Failure



Limitations of the pure master/slave topology

The pure master/slave topology is subject to the following limitations:

- Only one slave can be connected to the master.

- A failed master cannot be re-introduced without shutting down the the slave broker (no automatic failback).
- There is no automatic store synchronization between the brokers after a failure. Replicating the data from the slave back to the master is a manual process.

Configuring the master

No special configuration is required for the master, although there are a couple of optional attributes you can set on the master's `broker` element. Example 4.2 on page 67 shows a sample configuration for a master broker in the pure master/slave topology.

Example 4.2. Master Configuration for Pure Master/Slave

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

You can optionally set any of the following attributes on the `broker` element of a master broker:

`waitForSlave`
: If `true`, the master will wait untill a slave has attached before completing its start-up sequence. Default is `false`.

`shutdownOnSlaveFailure`
: If `true`, the master shuts down, if the slave connection is lost, thereby ensuring that the master does not get out of sync with the slave. Default is `false`.

Note

You should *not* configure a network connector between the master and its slave. Only the master connector should be created (in the slave's configuration). If you explicitly configure a network connector, you may encounter race conditions when the master broker is under heavy load.

Configuring the slave

There are two alternative approaches to configuring the slave. Example 4.3 on page 68 shows how to configure the slave using the `masterConnector` element, which enables you to specify username and password credentials for the connection.

Example 4.3. Slave Configuration for Pure Master/Slave

```
<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>
```

The `masterConnector` element supports the following attributes:

`remoteURI`
: Specifies the address of the master's transport connector port.

`userName`
: Username for connecting to the master (if authentication is enabled).

`password`
: Password for connecting to the master (if authentication is enabled).

Example 4.4 on page 68 shows the alternative approach to configuration, by setting attributes on the `broker` element. This approach does not support setting credentials, however.

Example 4.4. Alternative Slave Configuration

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

```
  </transportConnectors>
  ...
</broker>
```

The `masterConnector` element supports the following attributes:

`masterConnectorURI`
: Specifies the address of the master's transport connector port.

`shutdownOnMasterFailure`
: If `true`, the slave shuts down when the master fails; otherwise, the slave takes over as the new master. Default is `false`.

  The slave ensures that there is a separate copy of each message and acknowledgment on another machine, which can protect against catastrophic hardware failure. If the master fails, you might want the slave to shut down as well, if protection against data loss is your main priority. You can then manually duplicate the slave's copy of the data before restoring the system.

Recovery procedure

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

1. Shut down the slave broker.

Note

Clients do not need to be restarted. If they are failover clients, they will automatically reconnect when the master/slave topology is restored.

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

If you have an existing high availability (HA) infrastructure based on a shared file system (SAN file system), it is relatively easy to set up a fault-tolerant cluster of brokers using this technology. Brokers automatically configure themselves to operate in master mode or slave mode, depending on whether or not they manage to grab an exclusive lock on the underlying data directory. Replication of data is managed by the underlying SAN file system.

Storage area networks

A *storage area network (SAN)* is a storage system that enables you to attach remote storage devices (such as disk drives or tape drives) to a computer, making them appear as if they were local storage devices. A distinctive feature of the SAN architecture is that it combines data centres, physically separated by large distances, into a single storage network. With the addition of suitable software or hardware, a SAN system can be designed to provide data replication across multiple data centres, making it ideal for disaster recovery.

Because of the exceptional demands for high speed and reliability in a storage network (where gigabit bandwidths are required), SANs were originally built using dedicated fibre-optic or twisted copper wire cables and associated protocols, such as Fiber Channel (FC), were developed for this purpose. Alternative network protocols, such as FCoE and iSCSI, have also been developed to enable SANS to be built over high-speed Ethernet networks.

For more details about SANs, see the storage area network[1] Wikipedia article.

SAN filesystems

The SAN itself defines only block-level access to storage devices. Another layer of software, the *SAN file system* or *shared disk file system*, is needed to provide the file-level access, implementing the *file* and *directory* abstractions. Because the SAN file system can be shared between multiple computers, it must also be capable of regulating concurrent access to files. File locking is, therefore, an important feature of a SAN file system.

Exclusive file lock on a shared file system

A SAN file system must implement an efficient and reliable system of file locking to ensure that different computers cannot write to the same file at the same time. The shared file system master/slave failover pattern depends on a reliable file locking mechanism in order to function correctly.

[1] http://en.wikipedia.org/wiki/Storage_area_network

Warning

OCFS2 is incompatible with this failover pattern, because mutex file locking from Java is not supported.

Warning

NFSv3 is incompatible with this failover pattern. 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 the slave broker cannot acquire the lock and therefore cannot start up. In this case, the only way to unblock the failover cluster in NFSv3 is to reboot all broker instances.

On the other hand, NFSv4 *is* compatible with this failover pattern, 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.

Creating master and slave brokers

In the shared file system master/slave pattern, there is nothing special to distinguish a master broker from the slave brokers. There can be any number of brokers in a failover cluster. Membership of a particular failover cluster is defined by the fact that all of the brokers in the cluster use the *same* persistence layer and store their data in the *same* shared directory. The brokers in the cluster therefore compete to grab the exclusive lock on the data file. The first broker to grab the exclusive lock is the *master* and all of the other brokers in the cluster are the *slaves*. The master and the slaves now behave as follows:

- The master retains the exclusive lock on the data file, preventing the other brokers from accessing the data. The master starts up its transport connectors and network connectors, enabling other messaging clients and message brokers to connect to it.
- The slaves keep attempting to grab the lock on the data file, but they do not succeed as long as the master is running. The slaves do *not* start up any transport connectors or network connectors and are thus inaccessible to messaging clients and brokers.

Sample broker configuration

The only condition that brokers in a cluster must satisfy is that they all must use the *same* persistence layer and the persistence layer must put its data into a directory in a *shared file system*. For example, assuming

that `/sharedFileSystem/sharedBrokerData` is a directory in a shared file system, you could configure a Kaha DB persistence layer as follows:

Example 4.5. Shared file cluster configuration

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

Alternatively, the AMQ persistence layer is also suitable for this failover scenario:

Example 4.6. Alternate shared file cluster configuration

```
<persistenceAdapter>
  <amqPersistenceAdapter directory="/sharedFileSystem/sharedBrokerData"/>
</persistenceAdapter>
```

Sample client configuration

Clients of the failover cluster must be configured with a failover URL that lists the URLs for all of the brokers in the cluster. For example, assuming that there are three brokers in the cluster, deployed on the hosts, `broker1`, `broker2`, and `broker3`, and all listening on IP port `61616`, you could use the following failover URL for the clients:

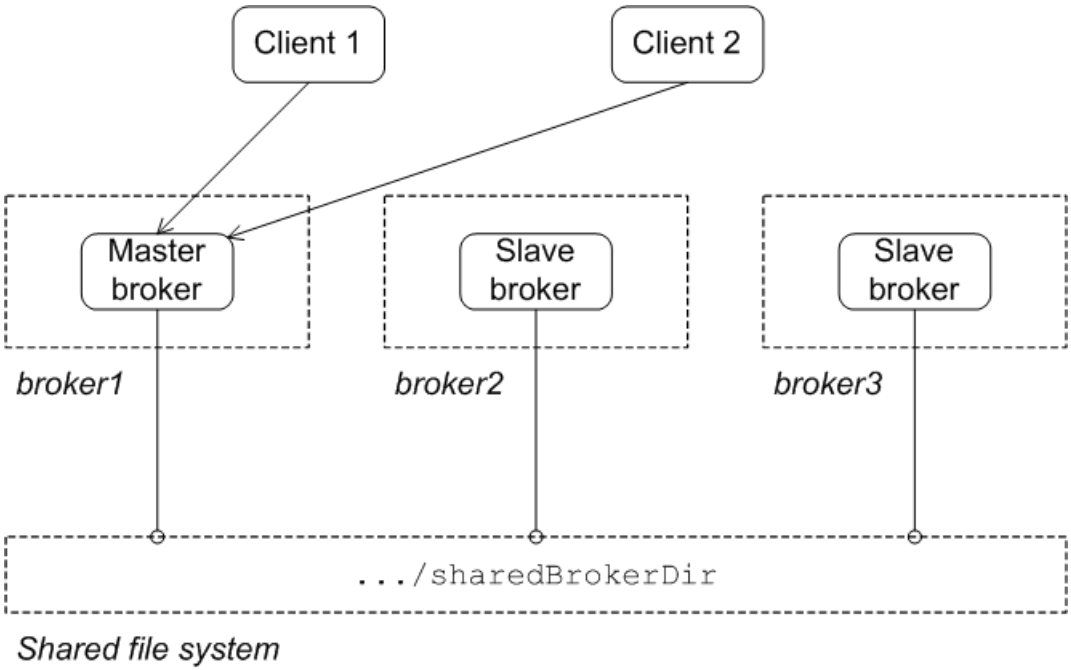
```
failover:(tcp://broker1:61616,tcp://broker2:61616,tcp://broker3:61616)
```

In this case, it does not matter in which order the clients attempt to connect to the brokers, because the identity of the master broker is determined by chance: that is, by whichever broker is the first to grab the exclusive lock on the data file.

Initial state

Figure 4.3 on page 73 shows the initial state of a shared file system master/slave cluster. When all of the brokers in the cluster are started, one of them grabs the exclusive lock on the broker data file, thus becoming the master. All of the other brokers in the clusters 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.

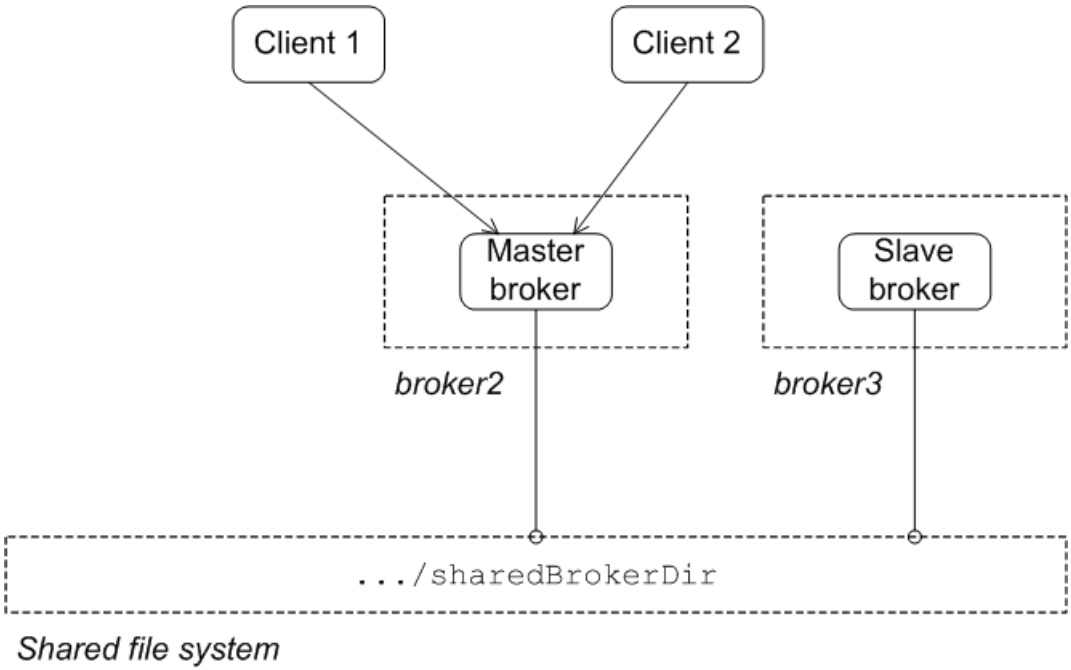
Figure 4.3. Shared File System Initial State



After failure of the master

Figure 4.4 on page 74 shows the state of the cluster 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 broker data file frees up and another broker in the cluster grabs the lock and gets promoted to master (`broker2` in the figure).

Figure 4.4. Shared File System after Master Failure

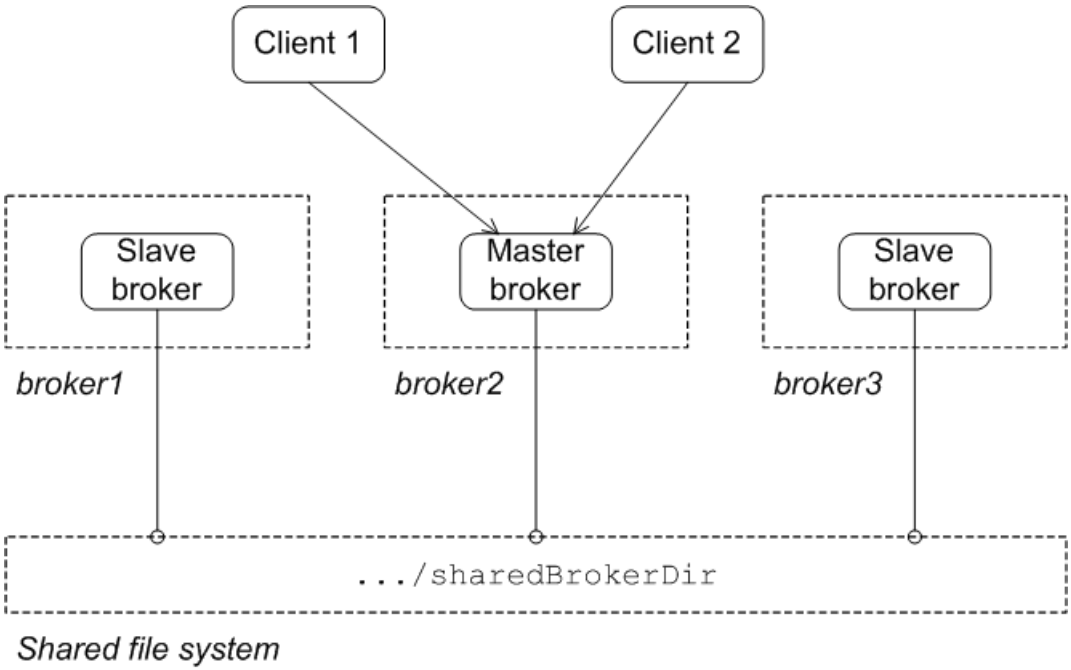


After the clients lose their connection to the original master, they automatically try all of the other brokers listed in the failover URL. This enables them to find and connect to the new master.

After restarting the failed master

You can restart the failed master at any time and it will rejoin the cluster. Initially, however, it will have the status of a slave broker, because one of the other brokers already owns the exclusive lock on the broker data file, as shown in Figure 4.5 on page 75.

Figure 4.5. Shared File System after Master Restart



JDBC Master/Slave

Overview

If it is acceptable for your application to have the database as a single point of failure, you can deploy a broker cluster using the JDBC master/slave failover pattern. Conceptually, this approach is somewhat similar to the shared file system master/slave pattern, as it is also based on lock acquisition. 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 an underlying database table. There is no replication of data in this scenario (unless this is provided by the database).

It is also necessary to access the database *without* enabling high speed journaling. This inevitably has a significant impact on performance and has to be kept in mind when comparing with other failover approaches.

Creating master and slave brokers

In the JDBC master/slave pattern, there is nothing special to distinguish a master broker from the slave brokers. Membership of a particular failover cluster is defined by the fact that all of the brokers in the cluster use the *same* JDBC persistence layer (without journaling) and store their data in the *same* database tables. The brokers in the cluster therefore compete to grab the mutex lock on the database table. The first broker to grab the lock is the *master* and all of the other brokers in the cluster are the *slaves* (there can be any number of brokers in a failover cluster). The master and the slaves now behave as follows:

- The master retains the lock on the database table, preventing the other brokers from accessing the data. The master starts up its transport connectors and network connectors, enabling other messaging clients and message brokers to connect to it.
- The slaves keep attempting to grab the lock on the database table, but they do not succeed as long as the master is running. The slaves do *not* start up any transport connectors or network connectors and are thus inaccessible to messaging clients and brokers.

Sample broker configuration

The only condition that brokers in a cluster must satisfy is that they all use the same non-journaling JDBC persistence layer with the broker data stored in the same underlying database tables.

For example, to store the shared broker data in an Oracle database, you could configure the non-journaled JDBC persistence layer, for all brokers in the cluster, as follows:

Example 4.7. 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>
```

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

Sample client configuration

Clients of the failover cluster must be configured with a failover URL that lists the URLs for all of the brokers in the cluster. For example, assuming that there are three brokers in the cluster, deployed on the hosts, `broker1`, `broker2`, and `broker3`, and all listening on IP port `61616`, you could use the following failover URL for the clients:

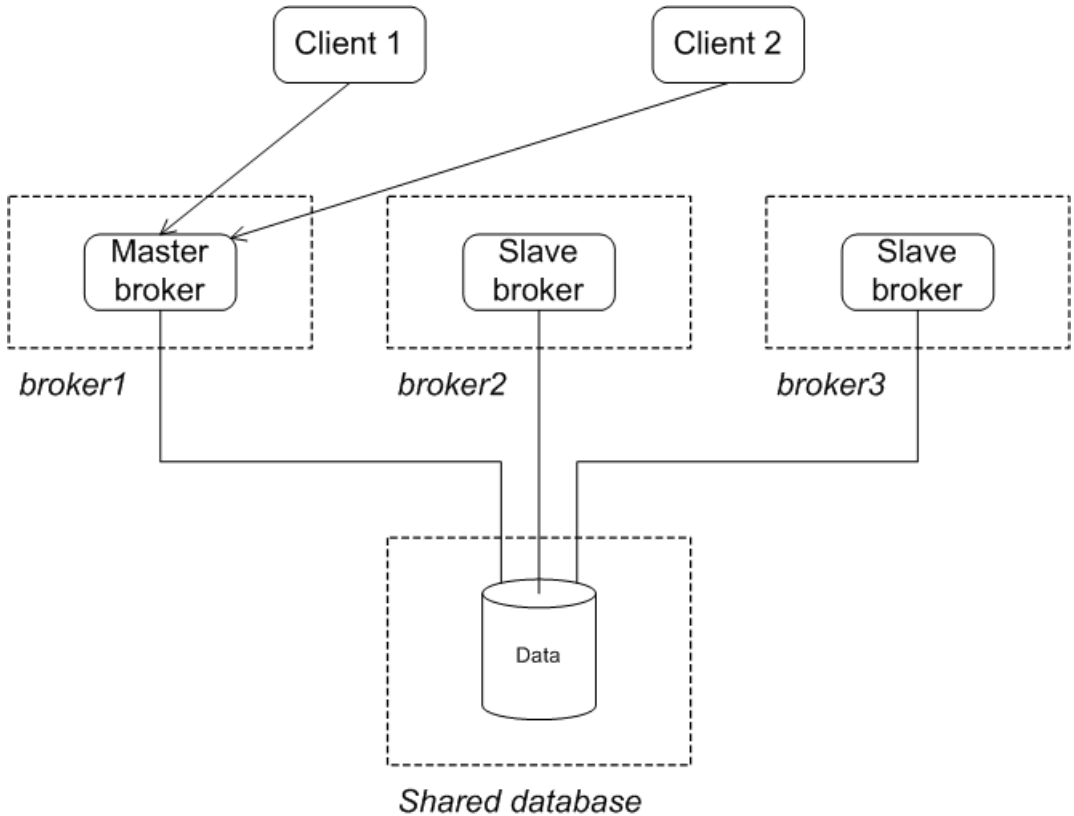
```
failover:(tcp://broker1:61616,tcp://broker2:61616,tcp://broker3:61616)
```

In this case, it does not matter in which order the clients attempt to connect to the brokers, because the identity of the master broker is determined by chance: that is, by whichever broker is the first to grab the mutex lock on the relevant database table.

Initial state

Figure 4.6 on page 78 shows the initial state of a JDBC master/slave cluster. When all of the brokers in the cluster are started, one of them grabs the mutex lock on the database table, thus becoming the master. All of the other brokers in the clusters remain 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 4.6. JDBC Master/Slave Initial State

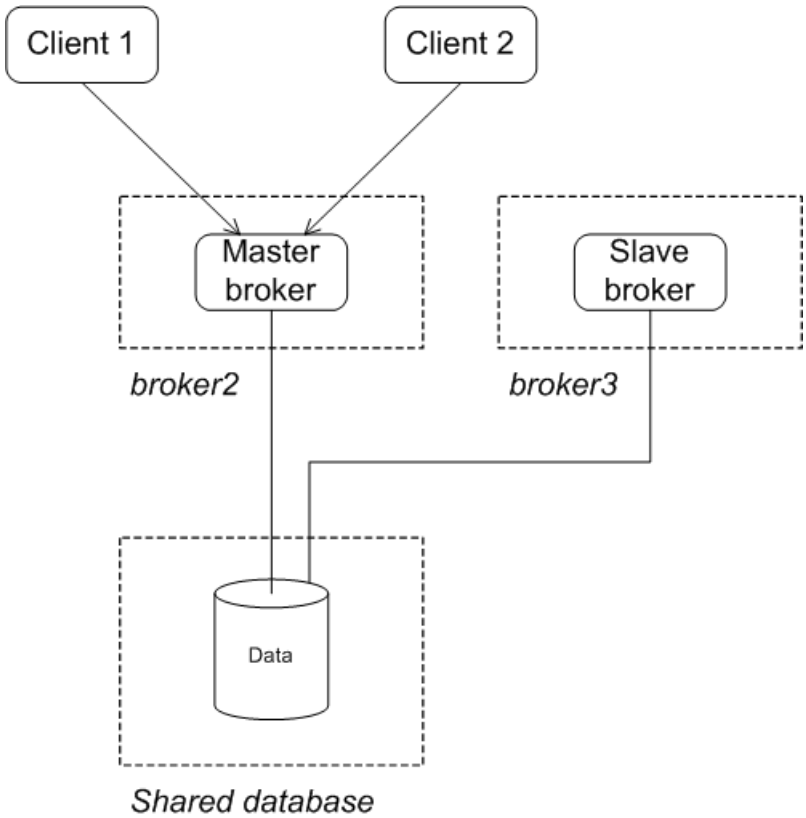


After failure of the master

Figure 4.7 on page 79 shows the state of the cluster 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 in the cluster grabs the lock and gets promoted to master (`broker2` in the figure).

Figure 4.7. JDBC Master/Slave after Master Failure

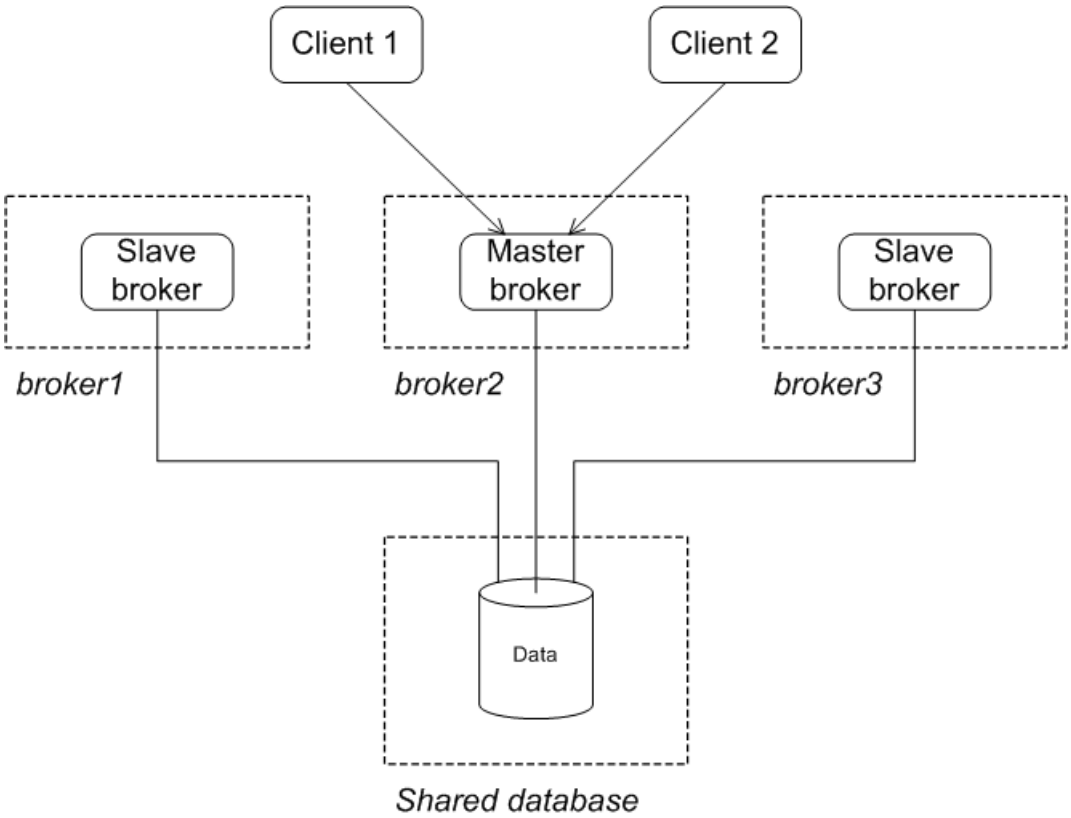


After the clients lose their connection to the original master, they automatically try all of the other brokers listed in the failover URL. This enables them to find and connect to the new master.

After restarting the failed master

You can restart the failed master at any time and it will rejoin the cluster. Initially, however, it will have the status of a slave broker, because one of the other brokers already owns the mutex lock on the database table, as shown in Figure 4.8 on page 80.

Figure 4.8. JDBC Master/Slave after Master Restart



Fault Tolerant Broker Network

Overview

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

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

- Network connectors to a master/slave cluster use a very specific configuration of the failover protocol.
- A broker cannot open a network connection to another member of its master/slave cluster.

Configuring the connection to a master/slave cluster

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

- Open a connection to the master broker in the master/slave cluster 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 failover URI to specify the list of broker's in the master/slave cluster. The failover URI only allows the connector to connect to one of brokers in the list which will be the master.

You *must* set the `maxReconnectAttempts=0` to disable the failover protocol's reconnect logic. Not doing so will cause problems recreating the network bridges when failing over to a new master broker.

Example 4.8 on page 81 shows a network connector configured to link to a master/slave cluster.

Example 4.8. Network Connector to a Master/Slave Cluster

```
<networkConnectors>
  <networkConnector name="linkToCluster"
                uri="static:failover:(tcp://masterHost:61002,tcp://slaveHost:61002)?maxRe
connectAttempts=0"
                ... />
</networkConnectors>
```

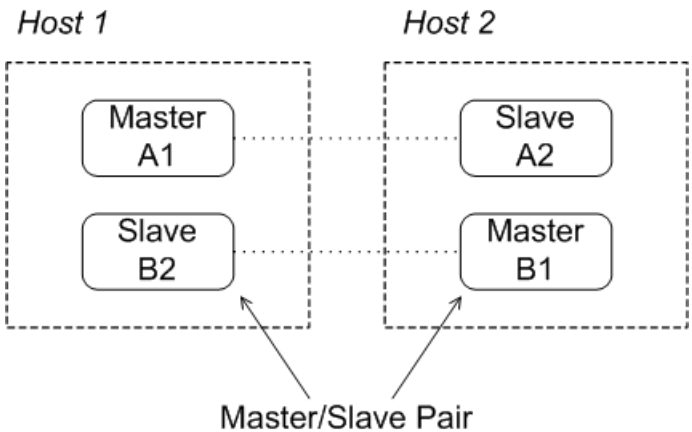
Important

`maxReconnectAttempts=0` is the only failover option you can use when configuring the network connector.

Host pair with master/slave pairs

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 4.9 on page 82.

Figure 4.9. Master/Slave Pairs on Two Host Machines



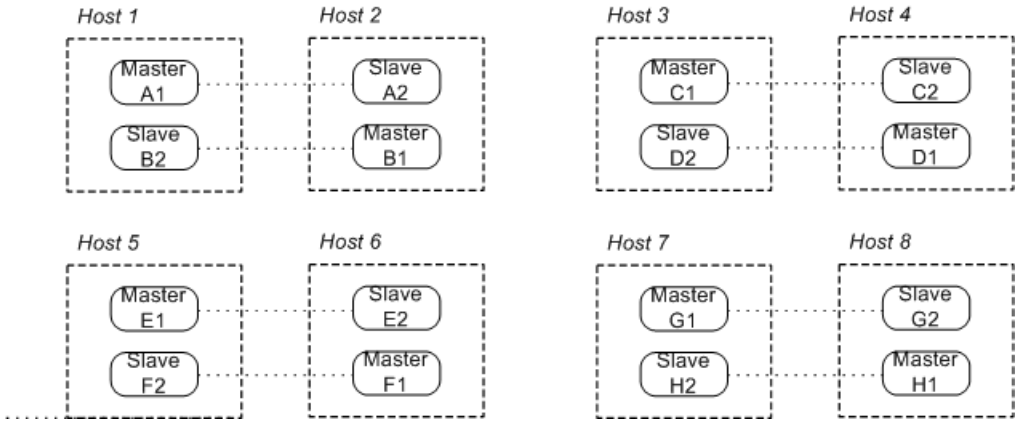
The host pair arrangement consists of two master/slave pairs distributed between two host machines. Under normal operating conditions, one master broker is active on each of the two host machines. If one of the machines should fail for some reason, the slave on the other machine takes over, so that you 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 failover cluster. 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 4.10 on page 83.

Figure 4.10. Broker Network Consisting of Host Pairs



The preceding network consists of eight fault tolerant pairs distributed over eight host machines. As before, you should open network connectors only to brokers outside the current failover cluster. For example, `brokerA1` can connect to at most the following brokers: `brokerB*`, `brokerC*`, `brokerD*`, `brokerE*`, `brokerF*`, `brokerG*`, and `brokerH*` (where `*` matches `1` or `2`).

Index