FuseSource

Fuse MQ Enterprise

Using Networks of Brokers

Version 7.1
December 2012

Integration Everywhere

Using Networks of Brokers

Version 7.1

Updated: 07 Jan 2014
Copyright © 2012 Red Hat, Inc. and/or its affiliates.

- 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)

- 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.

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

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

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

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

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction

Distributing your brokers can provide a number of benefits including fault tolerance, load balancing, and network segmentation. Fuse MQ Enterprise allows you to federate your brokers into a network of brokers so that distributed brokers can share information and route messages as needed.

Overview

For many applications, using a single message broker is sufficient. However, there are many cases where using multiple interconnected brokers is more appropriate. For example, if you need to ensure that your application is continuously available, if your application needs to process large volumes of messages, or if your integration solution calls for message processing across distributed location a network of brokers will work better than a single message broker.

Fuse MQ Enterprise facilitates these use cases by making it possible to build up a network of brokers. A network of brokers is a set of two or more brokers connected together by network connectors. All of the brokers in the network share information about the clients and destinations each broker hosts. The connected brokers use this information to route messages through the network.

Network of brokers

A network of brokers is created when one broker establishes a network connector to another broker. Once the network connector is established the broker that established the connection discovers information about the destinations being hosted on the other broker and which consumers are actively listening for messages on the destinations. Using this information, the first broker can route messages from its producers to consumers on the connected broker. A simple network of brokers, such as this, spreads load between the two brokers, allows each broker to be configured for specific needs, and partitions the producers and consumers.

A network of brokers can be expanded by introducing more brokers to the network. This allows you to build up sophisticated network topologies. You can also create bidirectional connections between brokers to allow for more sophisticated message routing.

Dynamic networks

To create a robust network, it is important to be able to deploy brokers dynamically through out your infrastructure. It is also important to be able to add and remove brokers as needed. Fuse MQ Enterprise facilitates this with a number of discovery protocols. These protocols allow brokers and clients to determine a list of active brokers. Brokers can automatically add new brokers to a network of brokers and removes inactive brokers. Clients always have a list of brokers that are available if they need to failover to a new broker.

Chapter 2. Network Connectors

The network connector is the glue that binds a network of brokers. They are define the pathways between the brokers and are responsible for controlling how messages propagate throughout the network.

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 MQ Enterprise 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 21 for details.

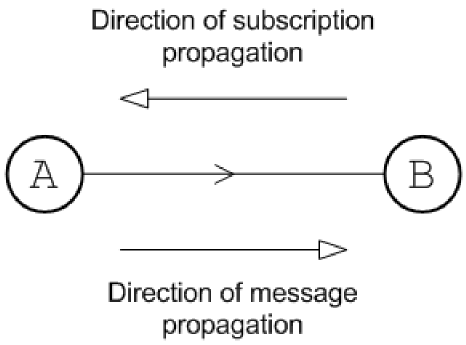
Network connector

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 15 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


Direction of subscription propagation
A
B
Direction of message propagation

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 52 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.

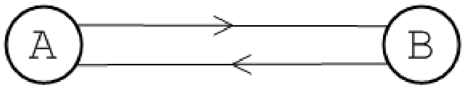
`networkTTL`

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.

Connectors in each direction

Figure 2.1 on page 15 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

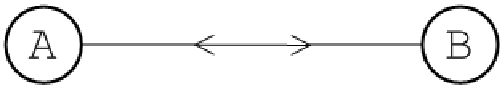
```
<beans ...>
    <broker xmlns="http://activemq.apache.org/schema/core"
            brokerName="brokerB" brokerId="B"... >
        ...
        <networkConnectors>
```

```
            <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 17 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.

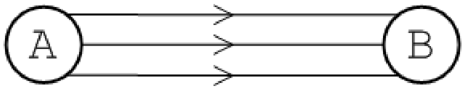
Tip

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 18 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:

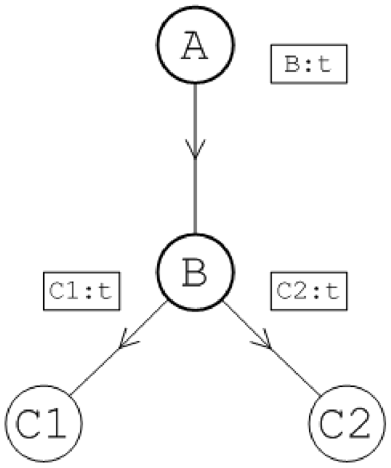
- Spreading the load amongst multiple connections.

- 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.

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 19 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 19 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 66 for more details.

Chapter 3. Dynamic and Static Propagation

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). However, there are cases in which messages need to be propagated in the absence of subscriptions.

Overview

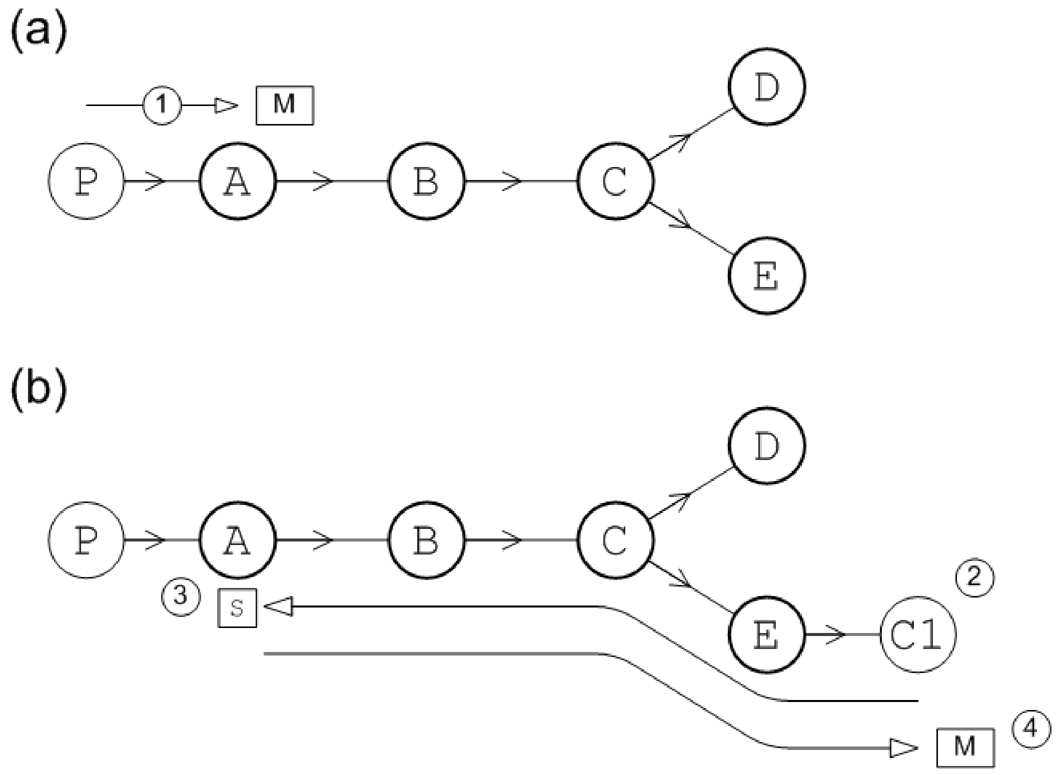
The fundamental purpose of a broker network is to route messages to their intended recipients, which are consumers that could be attached at any point in the network. The peculiar difficulty in devising routing rules for a messaging network is that messages are sent to an *abstract* destination rather 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 3.1 on page 22 illustrates how dynamic propagation works for messages sent to a queue. The broker connectors in this network are simple (non-duplex).

Figure 3.1. 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:

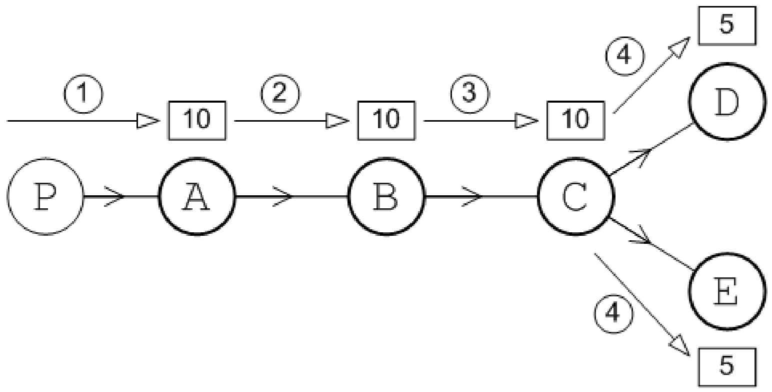
```
<networkConnectors>
    <networkConnector name="linkToBrokerB"
       uri="static:(tcp://localhost:61002)"
       networkTTL="3">
        <staticallyIncludedDestinations>
            <queue physicalName="TEST.FOO"/>
        </staticallyIncludedDestinations>
    </networkConnector>
</networkConnectors>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

Note

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

Consider the network shown in Figure 3.2 on page 24. 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 3.2. 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

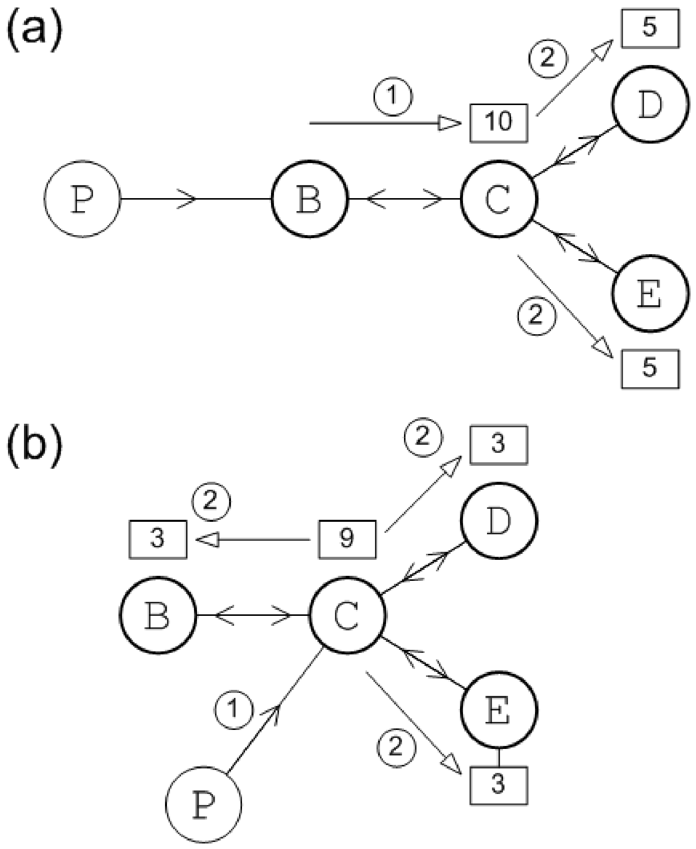
Using the preceding static configuration, it is possible for messages to get stuck in a particular broker. For example, if a consumer now connects to broker E, it will receive the 5 messages stored on broker

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 3.3 on page 25 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 3.3. 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:

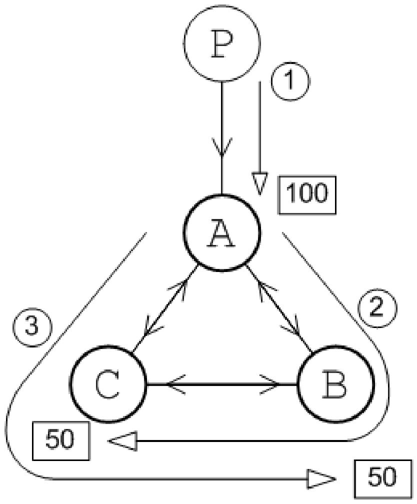
1. Because the duplex connector, `{B,C}`, has enabled static propagation for the queue, `TEST.FOO`, the 10 messages on broker B are forwarded to broker C.
2. Because the duplex connectors, `{C,D}` and `{C,E}`, have enabled static propagation for the queue, `TEST.FOO`, the 10 messages on broker C are 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 3.4 on page 27 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 3.4. 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 MQ Enterprise 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.

Chapter 4. Destination Filtering

One reason to create a network of brokers is to partition message destinations to sub-domains of the network. Fuse MQ Enterprise can apply filters to destination names to prevent messages for a destination from passing through a network connector.

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 MQ Enterprise lets you define filters on network connectors in order to control the flow of messages throughout the network.

Fuse MQ Enterprise allows you to control the flow of messages in two ways:

- specifying which destinations' messages can pass through a connector
- excluding messages for specific destinations from passing through a connector

Destination wildcards

Destination names are often segmented to denote how they are related. For example, an application may use the prefix `PRICE.STOCK` to denote all of the destinations that handle stock quotes. The application may then further segment the destination names such that all stock quotes from the New York Stock Exchange were prefixed with `PRICE.STOCK.NYSE` and stock quotes from NASDAQ used the prefix `PRICE.STOCK.NASDAQ`. Using wildcards would be a natural way to create filters for specific types of destinations.

Table 4.1 on page 29 describes the characters can be used to define wildcard matches for destination names.

Table 4.1. Destination Name Wildcards

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

Table 4.2 on page 30 shows some examples of destination wildcards and the names they would match.

Table 4.2. Example Destination Wildcards

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 destinations by inclusion

The default behavior of a network connector is to allow messages for all destinations to pass. You can, however, configure a network connector to only allow messages for specific destinations to pass. If you use segmented destination names, you can use wildcards to filter groups of destinations.

You do this by adding a `dynamicallyIncludedDestinations` child to the network connector's `networkConnector` element. The included destinations are specified using `queue` and `topic` children. Example 4.1 on page 30 shows configuration for a network connector that only passes messages destined for queues with names that match `TRADE.STOCK.>` and topics with names that match `PRICE.STOCK.>`.

Example 4.1. Network Connector Using Inclusive Filtering

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

Important

Once you add the `dynamicallyIncludedDestinations` to a network connector's configuration, the network connector will *only* pass messages for the specified destinations.

Filtering destinations by exclusion

Another way of partitioning a network and create filters is to explicitly specify a list destinations whose messages are not allowed to pass through a network connector. If you use segmented destination names, you can use wildcards to filter groups of destinations.

You do this by adding a `excludedDestinations` child to the network connector's `networkConnector` element. The excluded destinations are specified using `queue` and `topic` children. Example 4.2 on page 31 shows configuration for a network connector that blocks messages destined for queues with names that match `TRADE.STOCK.NYSE.*` and topics with names that match `PRICE.STOCK.NYSE.*`.

Example 4.2. Network Connector Using Exclusive Filtering

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

Combining inclusive and exclusive filters

You can combine inclusive and exclusive filtering to create complex network partitions. Example 4.3 on page 31 shows a network connector that is configured to transmit stock prices from any exchange except the NYSE and transmits orders to trade stocks for any exchange except the NYSE.

Example 4.3. Combining Exclusive and Inclusive Filters

```
<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>
```

Chapter 5. Using JMS Message Selectors

Fuse MQ Enterprise supports using JMS message selectors to filter messages. When using JMS message selectors with a network of brokers, you need to be aware of how the message selectors interact with conduit subscriptions. The interaction can lead to some undesirable outcomes if not properly managed.

Overview

JMS message selectors allow consumers to filter messages by testing the contents of a message's JMS header. The selectors are specified when the consumer connects to a broker and starts listing to messages on a particular destination. The broker then filters the messages that delivered to the consumer.

Brokers in a network also use JMS message selectors to determine how messages are routed. A consumer's message selectors are included in the subscription information propagated throughout the network. All of the brokers can then use this information to filter messages before forwarding messages through a network connector.

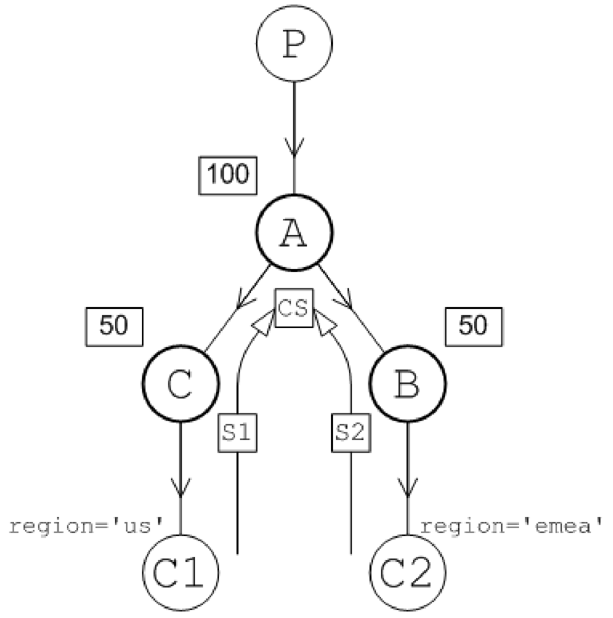
The one instance where message selectors are not used is when one or more consumer subscriptions are combined into a conduit subscription. This means that the broker receiving the conduit subscription cannot use the message selectors when determining what messages to forward.

Scenarios that do not work

Trouble arises when message selectors are combined with conduit subscriptions for consumers that are listening on the same queue.

Consider the broker network shown in Figure 5.1 on page 34. Consumers C1 and C2 subscribe to the same queue and they also define JMS message selectors. C1 selects messages for which the `region` header is equal to `us`. C2 selects messages for which the `region` header is equal to `emea`.

Figure 5.1. JMS Message Selectors and Conduit Subscriptions



The consumer subscriptions, `s1` and `s2`, automatically propagate to broker A. Because these subscriptions are both on the same queue broker A combines the subscriptions into a single conduit subscription, `cs`, which does *not* include any selector details. When the producer 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.

The best case scenario is that, by luck, the messages are forwarded to the broker with a selector that matches the message. The worst case scenario is 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. Chances are that the result would be somewhere in the middle. However, that means that at least some messages will sit at a broker where they will never be consumed.

If the consumers were both listening to a topic instead of a queue broker A would send a copy of every message to both networked brokers. All of the messages would get processed because C1 would consume the messages for the US region and C2 would consumer the messages for the EMEA region.

However, any messages for the EMEA region would sit unconsumed in broker C and any messages for the US region would sit unconsumed in broker B.

Resolving the problem

When you are faced with a network of brokers suffering from the effects of combining conduit subscriptions and message selectors and the consumers are listening to a queue, the easiest solution is to disable conduit subscriptions at the network connector where the problem arises.

You disable conduit subscriptions by setting the `networkConnector` element's `conduitSubscriptions` to `false`. Example 5.1 on page 35 shows configuration for a network connector with conduit subscriptions disabled.

Example 5.1. Disabling Conduit Subscriptions

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

If the problem arises using topics, the solution is more difficult. Disabling conduit subscriptions will cause more problems. In this case, you will need to rethink the requirements of your application. If you *must* use message selectors with topics in a network of brokers, you have two options:

- ensure that your network topology is such that messages won't be sent to brokers without appropriate consumers
- ensure that the orphaned messages will not create issues in your application

Chapter 6. Network Topologies

The topology of your network describes the pattern created by the pathways through your network. Different topologies are appropriate for particular use cases.

Overview

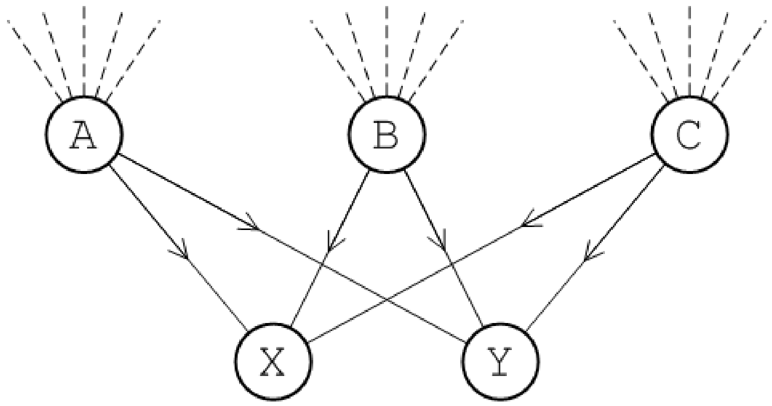
The following examples illustrate some of the common topologies encountered real-world networks:

- "Concentrator topology" on page 37.
- "Hub and spokes topology" on page 38.
- "Tree topology" on page 38.
- "Mesh topology" on page 39.
- "Complete graph" on page 40.

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 6.1 on page 37.

Figure 6.1. Concentrator Topology

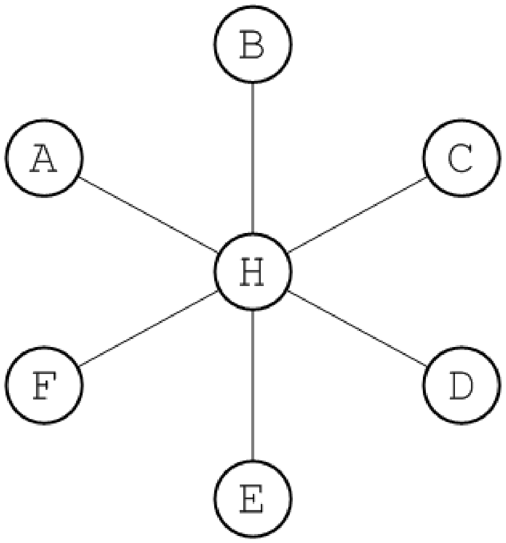


The idea of the concentrator topology is that you deploy brokers in two (or more) layers in order to funnel incoming connections into a smaller collection of services. The first layer consists of a relatively large number of brokers, with each broker servicing a large number of incoming connections (from 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 6.2 on page 38, 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 6.2. Hub and Spoke Topology

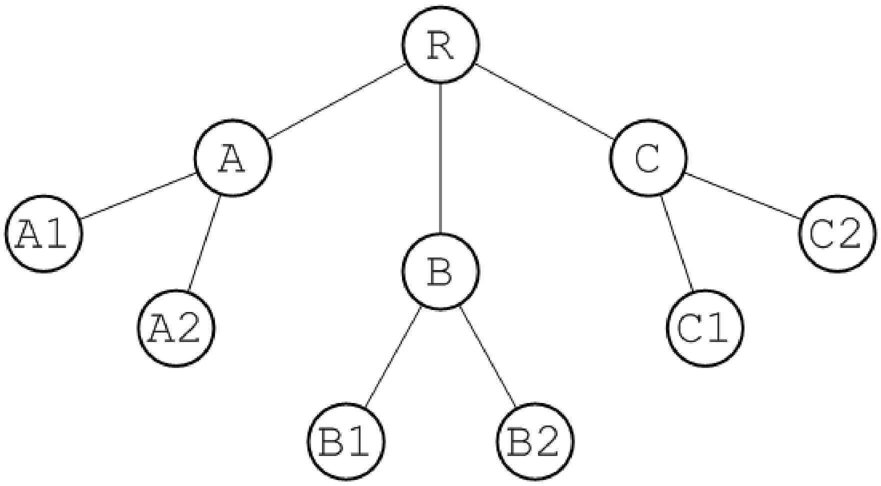


This topology is relatively robust. The only critical element is the hub node, so you would need to focus your maintenance efforts on keeping the hub up and running. Routes are determinate and the diameter of the network is always 2, no matter how many nodes are added.

Tree topology

The tree, as shown in Figure 6.3 on page 39, is a topology that arises naturally when a physical network grows in an informal manner.

Figure 6.3. Tree Topology

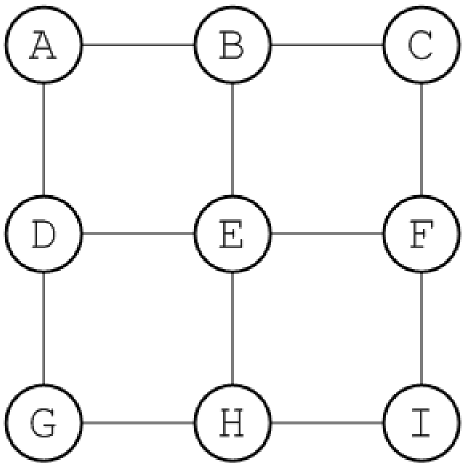


For example, if the network under consideration is an ethernet LAN, `R` could represent the hub in the basement of the IT department's building and `A` could represent a router in the ground floor of another building. If you want to extend the LAN to the first and second floor of building `A`, you are 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 6.4 on page 40, is a topology that arises naturally in a geographic network, when you decide to link together neighbouring hubs.

Figure 6.4. Mesh Topology

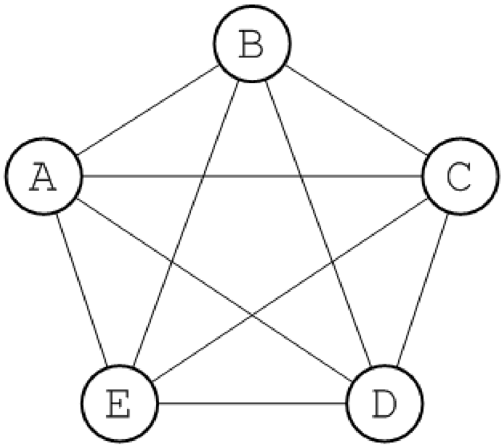


The diameter of a mesh increases whenever you add a node to its periphery. You must, therefore, be careful to set the network TTL sufficiently high that your network can cope with expansion. Alternatively, you could set up some mechanism for the central management of broker configurations. This would 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 6.5 on page 41 shows the graph, K_5.

Figure 6.5. 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 56).

Chapter 7. Optimizing Routes

It is possible, depending on your network's topology, that a message will multiple routes through the network. Fuse MQ Enterprise allows you to configure the network to reduce the number of alternate routes and choose the optimum route.

In network topologies such as a hub-and-spoke or a tree there exists a unique route between any two brokers. For topologies, such as a mesh or a complete graph, it is possible to have multiple routes between any two brokers. In such cases, you may need simplify the routing behavior, so that an optimum route is preferred by the network.

Fuse MQ Enterprise provides two configuration settings that work in conjunction to refine routing behavior:

- `decreaseNetworkConsumerPriority`—deprecates the priority of a network connector based on the number of hops from the message's origin so that messages are routed along the shortest route
- `suppressDuplicateQueueSubscriptions`—suppresses duplicate subscriptions from intermediary brokers so that alternative paths are reduced

Tip

To be most effective these properties should be set on *all* of the network connectors in the network of brokers.

Choosing the Shortest Route

Overview

In indeterminate networks, it is typically preferable for messages to take the *shortest* route. This reduces the time for the message to reach its destination, reduces the chances of the message being caught in a broker failure, and reduces the load on the network. In general, sending messages along to the nearest possible consumer maximizes the effectiveness of the broker network.

This is accomplished by configuring all of the connectors in your network to generate route priorities that automatically lowers the route's priority for each network connector it must traverse In this way the broker's can determine the shortest route between a message's producer and its consumer. In most cases, the broker will use the shortest route. However, if the shortest route is under heavy load, the broker will divert it to the next shortest route.

Connector configuration

To ensure that the shortest route is preferred, you need to configure *all* of the network connectors in the network to create priority profiles for each of the possible routes through the network. This is done by setting the `networkConnetor` element's `decreaseNetworkConsumerPriority` attribute to `true`.

Example 7.1 on page 44 shows a network connector configured to determine the shortest route.

Example 7.1. Network Connector for Choosing the Shortest Route

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

When `decreaseNetworkConsumerPriority` is set to `true`, the route priority is determined 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.

Important

If you choose not to enable `decreaseNetworkConsumerPriority` on all of the connectors in your network, the brokers will not be able to accurately determine the shortest route. Some network connectors will not have the proper starting priority and will not reduce their priority as required.

Route priority and broker load

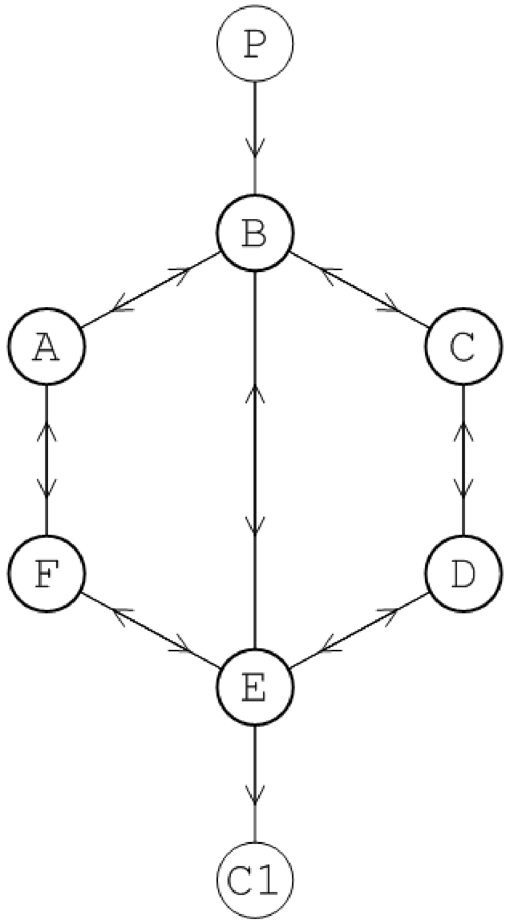
A broker prefers to send messages to the subscription with the highest priority. However, if the prefetch buffer for that subscription is full, the broker will divert messages to the subscription with the next highest priority.

If multiple subscriptions have the same priority, the broker distributes messages equally between those subscriptions.

Example

Figure 7.1 on page 46 illustrates the effect of activating `decreaseNetworkConsumerPriority` in a broker network.

Figure 7.1. Shortest Route in a Mesh Network



In this network, there are three alternative routes connecting producer P to consumer C1: `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 along this route unless connector `BE`'s prefetch buffer is full. In the case where connector `BE`'s prefetch buffer is full messages will be sent to route `PBAFEC1` and route `PBCDEC1` on an alternating basis.

Suppressing Duplicate Routes

Overview

Configuring your broker network to prefer the shortest route does not ensure that routing is deterministic. Under heavy load, the brokers will use the alternate routes to optimize performance. The danger of this is that if the message is routed along the longer alternate route and the consumer dies, the route becomes a dead-end and the message becomes stuck.

Fuse MQ Enterprise allows you to configure your network connectors to suppress duplicate subscriptions that arise from intermediary brokers. This has the effect of eliminating alternate paths between the networked brokers because only direct connections are recognized.

Connector configuration

To suppress duplicate subscriptions you set the `networkConnector` element's `suppressDuplicateQueueSubscriptions` attribute to `true` on all of the network connectors in you network. Example 7.2 on page 47 shows a network connector that is configured to suppress duplicate routes.

Example 7.2. Network Connector that Suppresses Duplicate Routes

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

Broker ID and duplicate routes

Fuse MQ Enterprise uses the brokers' IDs to figure out duplicate routes. In order for the suppression of duplicate routes to work reliably, you must give each broker a unique ID by explicitly setting the `broker` element's `brokerId` for each broker in the network. Example 7.3 on page 47 shows configuration setting a broker's ID.

Example 7.3. Setting a Broker's ID

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

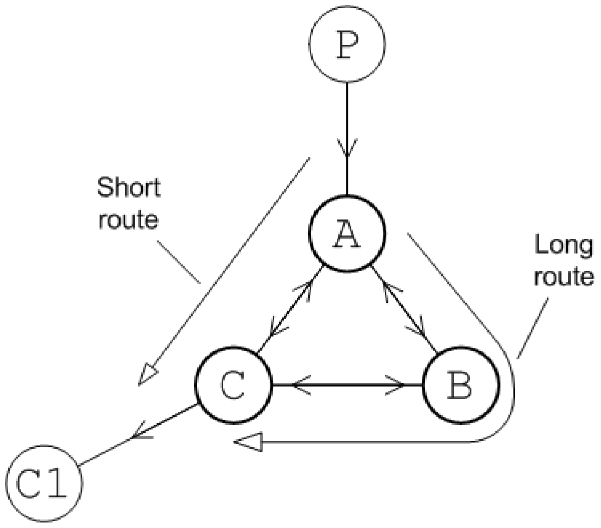
```
    ...
</broker>
```

Example

Consider the network of brokers, A, B, and C, shown in Figure 7.2 on page 48. In this scenario, a producer, P, connects to broker A and a consumer, C1 that subscribes to messages from P connects to broker B. The network TTL is equal to `2`, so two alternative routes are possible:

- the short route: `PABC1`
- long route: `PACBC1`

Figure 7.2. Duplicate Subscriptions in a Network



If you set `decreaseNetworkConsumerPriority` to `true`, the short route is preferred. and 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.

Setting `suppressDuplicateQueueSubscriptions` attribute to `true` will suppress the intermediary subscriptions that are generated between A and

B. Because this subscription is suppressed the only route left is `PACC1`. Routing becomes fully deterministic.

Note

In the example shown in Figure 7.2 on page 48, 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.

Chapter 8. Discovering Brokers

One of the main strengths of Fuse MQ Enterprise is that brokers can be located dynamically through out your infrastructure. In order for clients and other brokers to be able to interact with a broker, they need some way of discovering that the broker exists. Fuse MQ Enterprise does this using a combination of discovery agents and special URI schemes.

In order for location transparency to work, the members of a messaging application need a way for discovering each other. In Fuse MQ Enterprise this is accomplished using two pieces:

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

Discovery Agents

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

How a discovery agent learns about the available brokers varies between agents. Some agents use a static list, some use a third party registry, and some rely on the brokers to provide the information. For discovery agents that rely on the brokers for information, it is necessary to enable the discovery agent in the message broker configuration. For example, to enable the multicast discovery agent on an Openwire endpoint, you edit the relevant `transportConnector` element as shown in Example 8.1 on page 52.

Example 8.1. Enabling a Discovery Agent on a Broker

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

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

Tip

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

Fuse MQ Enterprise currently supports the following discovery agents:

- Fuse Fabric Discovery Agent

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

Fuse Fabric Discovery Agent

Overview

The *Fuse Fabric discovery agent* uses Fuse Fabric to discover the brokers in a specified group. The discovery agent requires that all of the discoverable brokers be deployed into a single fabric. When the client attempts to connect to a broker the agent looks up all of the available brokers in the fabric's registry and returns the ones in the specified group.

URI

The Fuse Fabric discovery agent URI conforms to the syntax in Example 8.2 on page 54.

Example 8.2. Fuse Fabric Discovery Agent URI Format

```
fabric://GID
```

Where *GID* is the ID of the broker group from which the client discovers the available brokers.

Configuring a broker

The Fuse Fabric discovery agent requires that the discoverable brokers are deployed into a single fabric.

The best way to deploy brokers into a fabric is using Fuse Management Console. For information on using Fuse Management Console see Fuse Management Console Documentation[1].

You can also use the console to deploy brokers into a fabric. See "Fabric Console Commands" in *Console Reference*.

Configuring a client

To use the agent a client must be configured to connect to a broker using a discovery protocol that uses a Fuse Fabric agent URI as shown in Example 8.3 on page 54.

Example 8.3. Client Connection URL using Fuse Fabric Discovery

```
discovery://(fabric://nwBrokers)
```

A client using the URL in Example 8.3 on page 54 will discover all the brokers in the `nwBrokers` broker group and generate a list of brokers to which it can connect.

[1] http://fusesource.com/docs/fmc

Static Discovery Agent

Overview

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

Using the agent

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

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

The static discovery agent URI conforms to the syntax in Example 8.4 on page 55.

Example 8.4. Static Discovery Agent URI Format

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

Example

Example 8.5 on page 55 shows a discovery URI that configures a client to use the static discovery agent to connect to one member of a broker pair.

Example 8.5. Discovery URI using the Static Discovery Agent

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

Multicast Discovery Agent

Overview

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

Important

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

URI

The multicast discovery agent URI conforms to the syntax in Example 8.6 on page 56.

Example 8.6. Multicast Discovery Agent URI Format

```
multicast://GroupID
```

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

Configuring a broker

For a broker to be discoverable using the multicast discovery agent, you must enable the discovery agent in the broker's configuration. To enable the multicast discovery agent you set the `transportConnector` element's `discoveryUri` attribute to a mulitcast discovery agent URI as shown in Example 8.7 on page 56.

Example 8.7. Enabling a Multicast Discovery Agent on a Broker

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

The broker configured in Example 8.7 on page 56 is discoverable as part of the multicast group `default`.

Configuring a client

To use the multicast agent a client must be configured to connect to a broker using a discovery URI that uses a multicast agent URI as shown in Example 8.8 on page 57.

Example 8.8. Client Connection URL using Multicast Discovery

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

A client using the URI in Example 8.8 on page 57 will discover all the brokers advertised in the `default` multicast group and generate a list of brokers to which it can connect.

Zeroconf Discovery Agent

Overview

The *zeroconf discovery agent* is derived from Apple's Bonjour Networking[2] technology, which defines the zeroconf protocol as a mechanism for discovering services on a network. Fuse MQ Enterprise bases its implementation of the zeroconf discovery agent on JmDSN[3], which is a service discovery protocol that is layered over IP/multicast and is compatible with Apple Bonjour.

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

Important

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

URI

The zeroconf discovery agent URI conforms to the syntax in Example 8.9 on page 58.

Example 8.9. Zeroconf Discovery Agent URI Format

```
zeroconf://GroupID
```

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

Configuring a broker

For a broker to be discoverable using the zeroconf discovery agent, you must enable a multicast discovery agent in the broker's configuration. To enable the multicast discovery agent you set the `transportConnector` element's `discoveryUri` attribute to a mulitcast discovery agent URI as shown in Example 8.10 on page 59.

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

Example 8.10. Enabling a Multicast Discovery Agent on a Broker

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

The broker configured in Example 8.10 on page 59 is discoverable as part of the multicast group `NEGroup`.

Configuring a client

To use the agent a client must be configured to connect to a broker using a discovery protocol that uses a zeroconf agent URI as shown in Example 8.11 on page 59.

Example 8.11. Client Connection URL using Zeroconf Discovery

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

A client using the URL in Example 8.11 on page 59 will discover all the brokers advertised in the `NEGroup` multicast group and generate a list of brokers to which it can connect.

Dynamic Discovery Protocol

Overview

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

URI syntax

Example 8.12 on page 60 shows the syntax for a discovery URI.

Example 8.12. Dynamic Discovery URI

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

`DiscoveryAgentUri` is URI for the discovery agent used to build up the list of available brokers. Discovery agents are described in "Discovery Agents" on page 52.

The options, `?Options`, are specified in the form of a query list. The discovery options are described in Table 8.1 on page 60. You can also inject transport options as described in "Setting options on the discovered transports" on page 61.

Tip

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

Transport options

The discovery protocol supports the options described in Table 8.1 on page 60.

Table 8.1. Dynamic Discovery Protocol Options

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

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

Sample URI

Example 8.13 on page 61 shows a discovery URI that uses a multicast discovery agent.

Example 8.13. Discovery Protocol URI

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

Setting options on the discovered transports

The list of transport options, *`Options`*, in the discovery URI can also be used to set options on the *discovered* transports. If you set an option *not* listed in "Setting options on the discovered transports" on page 61, the URI parser attempts to inject the option setting into every one of the discovered endpoints.

Example 8.14 on page 61 shows a discovery URI that sets the TCP `connectionTimeout` option to 10 seconds.

Example 8.14. Injecting Transport Options into a Discovered Transport

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

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

Fanout Protocol

Overview

The *fanout protocol* enables a producer to auto-discover broker endpoints and broadcast topic messages to *all* of the discovered brokers. The fanout protocol gives producers a convenient mechanism for broadcasting messages to multiple brokers that are not part of a network of brokers.

The fanout protocol relies on a discovery agent to build up the list of broker URIs to which it connects.

URI syntax

Example 8.15 on page 62 shows the syntax for a fanout URI.

Example 8.15. Fanout URI Syntax

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

`DiscoveryAgentUri` is URI for the discovery agent used to build up the list of available brokers. Discovery agents are described in "Discovery Agents" on page 52.

The options, `?Options`, are specified in the form of a query list. The discovery options are described in Table 8.2 on page 62. You can also inject transport options as described in "Setting options on the discovered transports" on page 61.

Tip

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

Transport options

The fanout protocol supports the transport options described in Table 8.2 on page 62.

Table 8.2. Fanout Protocol Options

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

Option Name	Default	Description
`maxReconnectDelay`	`30000`	Specifies, in milliseconds, the maximum amount of time to wait between reconnect attempts.
`useExponentialBackOff`	`true`	Specifies if an exponential back-off is used between reconnect attempts.
`backOffMultiplier`	`2`	Specifies the exponent used in the exponential back-off algorithm.
`maxReconnectAttempts`	`0`	Specifies the maximum number of reconnect attempts before an error is sent back to the client. `0` specifies unlimited attempts.
`fanOutQueues`	`false`	Specifies whether queue messages are replicated to every connected broker. For more information see "Applying fanout to queue messages" on page 63.
`minAckCount`	`2`	Specifies the minimum number of brokers to which the client must connect before it sends out messages. For more informaiton see "Minimum number of brokers" on page 64.

Sample URI

Example 8.16 on page 63 shows a discovery URI that uses a multicast discovery agent.

Example 8.16. Fanout Protocol URI

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

Applying fanout to queue messages

The fanout protocol replicates topic messages by sending each topic message to all of the connected brokers. By default, however, the fanout protocol does *not* replicate queue messages.

For queue messages, the fanout protocol picks one of the brokers at random and sends all of the queue messages to that broker. This is a sensible default, because under normal circumstances, you would not want to create more than one copy of a queue message.

It is possible to change the default behavior by setting the `fanOutQueues` option to `true`. This configures the protocol so that it also replicates queue messages.

Minimum number of brokers

By default, the fanout protocol does not start sending messages until the producer has connected to a *minimum of two brokers*. You can customize this minimum value using the `minAckCount` option.

Setting minimum number of brokers equal to the expected number of discovered brokers ensures that all of the available brokers start receiving messages at the same time. This ensures that no messages are missed if a broker starts up after the producer has started sending messages.

Using fanout with a broker network

You have to be careful when using the fanout protocol with brokers that are joined in a network of brokers.

The combination of the fanout protocol's broadcasting behavior and the nature of how messages are propagated through a network of brokers makes it likely that consumers will receive duplicate messages. If, for example, you joined four brokers into a network of brokers and connected a consumer listening for messages on topic `hello.jason` to broker A and connected a producer to broker B to send messages to topic `hello.jason`, the consumer would get one copy of the messages. If, on the other hand, the producer connects to the network using the fanout protocol, the producer will connect to every broker in the network simultaneously and start sending messages. Each of the four brokers will receive a copy of every message and deliver its copy to the consumer. So, for each message, the consumer will get four copies.

Chapter 9. Load Balancing

Broker networks can address the problem of load balancing in a messaging system. Consumer load is managed by changing how network connectors recognize subscriptions. Producer load is managed using different broker topologies.

Balancing Consumer Load

Overview

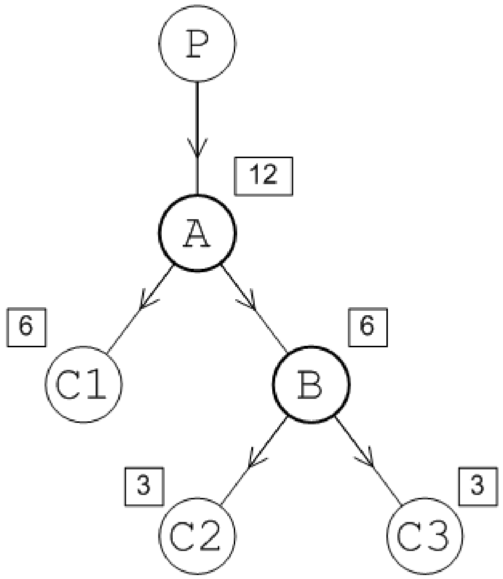
Multiple consumers attached to a JMS queue automatically obey *competing consumer* semantics. That is, each message transmitted by the queue is consumed by *one consumer only*. Hence, if you want to scale up load balancing on the consumer side, all that you need to do is attach extra consumers to the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

The default behavior of Fuse MQ Enterprise's conduit subscriptions, however, can sometimes be detrimental to load balancing on the consumer side. As described in "Conduit subscriptions" on page 19, conduit subscriptions concentrate all of the subscriptions from a networked broker into a single subscription. For topics this behavior optimizes traffic and has no effect on consumer load. For queues, however, it results in uneven message distribution which can impede consumer load balancing.

Default load behavior

Figure 9.1 on page 67 illustrates how conduit subscriptions can result in uneven message distribution to the consumers of a queue.

Figure 9.1. Message Flow when Conduit Subscriptions Enabled



Assume that the consumers, C1, C2, and C3, all subscribe to the `TEST.FOO` queue. Producer, P, connects to Broker A and sends 12 messages to the `TEST.FOO` queue. By default conduit subscriptions are enabled and Broker A sees only a single subscription from Broker B and a single subscription from consumer C1. So, Broker A sends messages alternately to C1 and B. Assuming that C1 and B process messages at the same speed, A sends a total of 6 messages to C1 and 6 messages to B.

Broker B sees two subscriptions, from C2 and C3 respectively. So, Broker B will send messages alternately to C2 and C3. Assuming that both consumers process messages at equal speed, each consumer receives a total of 3 messages.

In the end, the distribution of messages amongst the consumers is 6, 3, 3, which is not optimally load balanced. C1 processes twice as many messages as either C2 or C3.

Disabling conduit subscriptions

If you want to improve the load balancing behavior for queues, you can disable conduit subscriptions by setting the `networkConnector` element's `conduitSubscriptions` to `false`. Example 9.1 on page 68 shows configuration for a network connector with conduit subscriptions disabled.

Example 9.1. Disabling Conduit Subscriptions

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

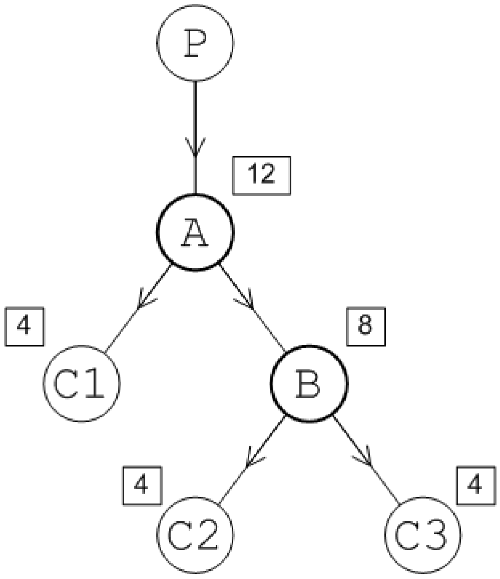
Warning

As described in "Conduit subscriptions" on page 19, conduit subscriptions protect against duplicate topic messages. If you are using both queues and topics consider using separate network connectors for queues and topics. See "Separate connectors for topics and queues" on page 69.

Balanced load behavior

Figure 9.2 on page 68 illustrates the message flow through a queue with distributed consumers when conduit subscriptions are disabled.

Figure 9.2. Message Flow when Conduit Subscriptions Disabled



Assume that the consumers, C1, C2, and C3, all subscribe to the `TEST.FOO` queue. Producer, P, connects to Broker A and sends 12 messages to the `TEST.FOO` queue. With conduit subscriptions disabled, Broker A sees both of the subscriptions on Broker B and a single subscription from consumer C1. Broker A sends messages alternately to each of the subscriptions. Assuming that all of the consumers process messages at equal speeds, C1 receives 4 messages and Broker B receives 8 messages.

Broker B sees two subscriptions, from C2 and C3 respectively. So, Broker B will send messages alternately to C2 and C3. Assuming that both consumers process messages at equal speed, each consumer receives a total of 4 messages.

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

Separate connectors for topics and queues

If your brokers need to handle both queues and topics, you might need to *disable* conduit subscriptions for queues to optimize load balancing, but also *enable* conduit subscriptions for topics to avoid duplicate topic messages.

Because the `conduitSubscriptions` attribute applies simultaneously to queues and topics, you cannot configure this using a single network connector. It is possible to configure topics and queues differently by using multiple network connectors: one for queues and another for topics.

Example 9.2 on page 69 shows how to configure separate network connectors for topics and queues. The `queuesOnly` network connector, which has conduit subscriptions disabled, is equipped with a filter that transmits only queue messages. The `topicsOnly` network connector, which has conduit subscriptions enabled, is equipped with a filter that transmits only topic messages.

Example 9.2. 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">
        <dynamicallyIncludedDestinations>
            <topic physicalName=">"/>
        </dynamicallyIncludedDestinations>
    </networkConnector>
</networkConnectors>
```

Managing Producer Load

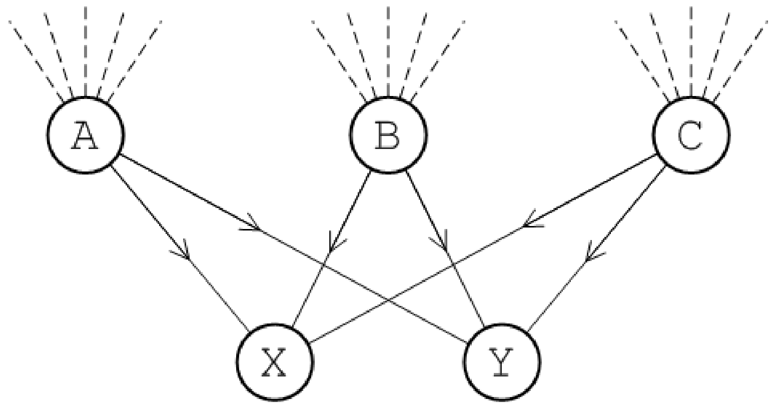
Overview

For greater scalability on the producer side, you might want to spread the message load across multiple brokers. For the purpose of spreading the load across brokers, one of the most useful topologies is the concentrator topology.

Concentrator topology

Figure 9.3 on page 71 illustrates a two layer network arranged in a concentrator topology.

Figure 9.3. Load Balancing with the Concentrator Topology



The two layers of brokers manage the producer load 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. Assuming the number of consumers is small, the brokers, X and Y, only need to deal with a relatively small number of connections. 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 laid out in a concentrator topology, producers and consumers must be configured to connect to the brokers in the appropriate layer. In the case of a producer connecting to the concentrator topology shown in Figure 9.3 on page 71, producers should connect to the brokers in the first layer: A, B, and C. Consumers should connect to the brokers in the second layer: X and Y.

Index