Red Hat AMQ 6.3

Fault Tolerant Messaging

Hardening your Red Hat JBoss A-MQ environment against downtime

Red Hat AMQ 6.3 Fault Tolerant Messaging

Hardening your Red Hat JBoss A-MQ environment against downtime

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

This guide shows you how to configure your broker to ensure maximum uptime and guarantee that messages are delivered.

Table of Contents

CHAPTER 1. INTRODUCTION

Abstract

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

OVERVIEW

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

- deploying multiple brokers into a topology that allows one broker to pick up the duties of a failed broker

- configuring clients to fail over to a new broker in the event that its current broker fails

Red Hat JBoss A-MQ provides mechanisms that make building fault tolerant messaging systems easy.

CLIENT FAIL OVER

JBoss A-MQ provides two protocols that allow clients to fail over to a new broker in the case of a failure:

- the failover protocol—allows you to provide a list of brokers that a client can use

- the discovery protocol—allows clients to automatically discover the brokers available for fail over

Both protocols automatically reconnect to an available broker when its existing connection fails. As long as an available broker is running, the client can continue to function uninterrupted.

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

For more information, see Chapter 2, *Client Failover*.

MASTER/SLAVE TOPOLOGIES

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

For details, see Chapter 3, *Master/Slave*.

CHAPTER 2. CLIENT FAILOVER

Abstract

Red Hat JBoss A-MQ provides two simple mechanisms for clients to failover to an alternate broker if its active connection fails. The failover protocol relies on a hard coded list of alternative brokers. The discovery protocol relies on discovery agents to provide a list of alternative brokers.

2.1. FAILOVER PROTOCOL

Abstract

The failover protocol provides a simple mechanism for clients to failover to a secondary broker if the primary connection fails. The failover protocol does not require that the brokers be configured as a network of brokers. However, when paired with a network of brokers it can be configured to use dynamic failover locations.

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

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

- Static—the client is configured with a static list of available URIs

- Dynamic—the brokers push information about the available broker connections

2.1.1. Static Failover

Overview

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

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

Failover URI

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

- failover:*uri1,...,uriN*

- failover:(*uri1,...,uriN*)?*TransportOptions*

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

Transport options

The failover protocol supports the transport options described in Table 2.1, "Failover Transport Options".

Table 2.1. Failover Transport Options

Option	Default	Description
initialReconnectDelay	10	Specifies the number of milliseconds to wait before the first reconnect attempt.
maxReconnectDelay	30000	Specifies the maximum amount of time, in milliseconds, to wait between reconnect attempts.
useExponentialBackOff	true	Specifies whether to use an exponential back-off between reconnect attempts.
backOffMultiplier	2	Specifies the exponent used in the exponential back-off algorithm.
maxReconnectAttempts	-1	Specifies the maximum number of reconnect attempts before an error is returned to the client. -1 specifies unlimited attempts. 0 specifies that an initial connection attempt is made at start-up, but no attempts to fail over to a secondary broker will be made.
startupMaxReconnectAttempts	-1	Specifies the maximum number of reconnect attempts before an error is returned to the client on the *first* attempt by the client to start a connection. -1 specifies unlimited attempts and 0 specifies no retry attempts.
randomize	true	Specifies if a URI is chosen at random from the list. Otherwise, the list is traversed from left to right.

Option	Default	Description
backup	false	Specifies if the protocol initializes and holds a second transport connection to enable fast failover.
timeout	–1	Specifies the amount of time, in milliseconds, to wait before sending an error if a new connection is not established. –1 specifies an infinite timeout value.
trackMessages	false	Specifies if the protocol keeps a cache of in-flight messages that are flushed to a broker on reconnect.
maxCacheSize	131072	Specifies the size, in bytes, used for the cache used to track messages.
updateURIsSupported	true	Specifies whether the client accepts updates to its list of known URIs from the connected broker. Setting this to false inhibits the client's ability to use dynamic failover. See Section 2.1.2, "Dynamic Failover".
updateURIsURL		Specifies a URL locating a text file that contains a comma-separated list of URIs to use for reconnect in the case of failure. See Section 2.1.2, "Dynamic Failover".
nested.*		Specifies extra options that can be added to the nested URLs. see Example 2.4, "Adding Options for Nested Failover URL"
warnAfterReconnectAttempts	10	Specifies to log a warning for no connection after every N number of attemps to reconnect. To disable the feature, set the value to <=0

Example

Example 2.1, "Simple Failover URI" shows a failover URI that can connect to one of two message brokers.

> Example 2.1. Simple Failover URI

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

2.1.2. Dynamic Failover

Abstract

Dynamic failover combines the failover protocol and a network of brokers to allow a broker to supply its clients with a list of broker connections to which the clients can failover.

Overview

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

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

Set-up

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

- The client's must be configured to use the failover protocol when connecting with its broker.

- The client must be configured to accept URI lists from a broker.

- The brokers must be configured to form a network of brokers.

  See "Using Networks of Brokers".

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

Client-side configuration

The client-side configuration for using dynamic failover is nearly identical to the client-side configuration for using static failover. The differences include:

- The failover URI can consist of a single broker URI.

- The **updateURIsSupported** option must be set to **true**.

- The **updateURIsURL** option should be set so that the transport can failover to a new broker when none of the broker's in the dynamically supplied list are available.

See the section called "Failover URI" and the section called "Transport options" for more information about using failover URIs.

Broker-side configuration

IMPORTANT

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

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

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

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

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

Table 2.2, "Broker-side Failover Properties" describes the broker-side properties that can be used to configure a failover cluster. These properties are attributes on the broker's **transportConnector** element.

Table 2.2. Broker-side Failover Properties

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

Example

Example 2.2, "Broker for Dynamic Failover" shows the configuration for a broker that participates in dynamic failover.

Example 2.2. Broker for Dynamic Failover

```
<beans ... >
  <broker>
    ...
    <networkConnectors>
1       <networkConnector uri="multicast://default" />
    </networkConnectors>
    ...
    <transportConnectors>
      <transportConnector name="openwire"
uri="tcp://0.0.0.0:61616"
2         discoveryUri="multicast://default"
3           updateClusterClients="true"
4             updateClusterFilter="A.*,B.*" />
      </transportConnectors>
    ...
  </broker>
</beans>
```

The configuration in Example 2.2, "Broker for Dynamic Failover" does the following:

1 Creates a network connector that connects to any discoverable broker that uses the multicast transport.

2 Makes the broker discoverable by other brokers over the multicast protocol.

3 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.

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

Example 2.3, "Failover URI for Connecting to a Failover Cluster" shows the URI for a client that uses the failover protocol to connect to the broker and its cluster.

Example 2.3. Failover URI for Connecting to a Failover Cluster

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

Example 2.4, "Adding Options for Nested Failover URL" shows the options that are passed to a nested URL to detect dead connections in the failover protocol.

Example 2.4. Adding Options for Nested Failover URL

```
failover:(tcp://host01:61616,tcp://host02:61616,tcp://host03:61616)?
nested.wireFormat.maxInactivityDuration=1000
```

2.1.3. Priority Backup

The priority backup feature allows clients to automatically reconnect to local urls. If the setup has brokers in both local and remote networks, you would want the client to connect to the local broker. In this scenario the priority backup feature is useful.

Example 2.5. Priority backup

```
failover:(tcp://local:61616,tcp://remote:61616)?randomize=false&priorityBackup=true
```

If the above url is used for the client, the client tries to connect to the local broker. If local broker fails, the client will fail over to the remote one. But as the **priorityBackup** parameter is used, the client will constantly try to reconnect to the local broker and the client switches to the local broker without any intervention.

Example 2.6. Priority URI

In case of multiple local urls the priority is assigned to the url by using **priorityURIs** parameter

```
failover:(tcp://local1:61616,tcp://local2:61616,tcp://remote:61616)?
randomize=false&priorityBackup=true&priorityURIs=tcp://local1:61616,tcp://local2:61616
```

The client prioritizes the brokers in order (first local1 then local2) and reconnects to them in that order according to their availability.

2.2. DISCOVERY PROTOCOL

Abstract

When you want to have the list of available brokers dynamically generated and don't want to hard code the initial connection point, you can use Red Hat JBoss A-MQ's discovery protocol. This protocol uses discovery agents to advertise available brokers and discovery URIs to configure what discovery agent a client will use for discovering brokers.

The failover protocol provides a lot of control over the brokers to which a client can connect. Using dynamic failover adds some ability to make the broker list more transparent. However, it has weaknesses. It requires that you know the address of at least one broker and that an initial broker is active when the client starts up. Using dynamic failover also requires that all of the brokers being used for failover are configured in a network of brokers.

Red Hat JBoss A-MQ's discovery protocol offers an alternative method for dynamically generating a list of brokers that are available for client failover. The protocol allows brokers to advertise their availability and for clients to dynamically discover them. This is accomplished using two pieces:

- *discovery URI*—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*—components that advertise the list of available brokers

2.2.1. Dynamic Discovery Protocol

Abstract

The dynamic discovery protocol combines reconnect logic with a discovery agent to dynamically create a list of brokers to which the client can connect.

Overview

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

URI syntax

Example 2.7, "Dynamic Discovery URI" shows the syntax for a discovery URI.

> **Example 2.7. 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 Section 2.2.2, "Discovery Agents".

The options, **?***Options*, are specified in the form of a query list. The discovery options are described in Table 2.3, "Dynamic Discovery Protocol Options". You can also inject transport options as described in the section called "Setting options on the discovered transports".

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

Transport options

The discovery protocol supports the options described in Table 2.3, "Dynamic Discovery Protocol Options".

Table 2.3. Dynamic Discovery Protocol Options

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

Option	Default	Description
maxReconnectDelay	**30000**	Specifies, in milliseconds, the maximum amount of time to wait between reconnect attempts.
useExponentialBackOff	**true**	Specifies if an exponential back-off is used between reconnect attempts.
backOffMultiplier	**2**	Specifies the exponent used in the exponential back-off algorithm.
maxReconnectAttempts	**-1**	Specifies the maximum number of reconnect attempts before an error is sent back to the client. **-1** specifies unlimited attempts. **0** denotes that reconnects are disabled, i.e., try once to reconnect. Values greater than **0** denote the maximum number of reconnect attempts.

Sample URI

Example 2.8, "Discovery Protocol URI" shows a discovery URI that uses a multicast discovery agent.

Example 2.8. 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 the section called "Setting options on the discovered transports", the URI parser attempts to inject the option setting into every one of the discovered endpoints.

Example 2.9, "Injecting Transport Options into a Discovered Transport" shows a discovery URI that sets the TCP **connectionTimeout** option to 10 seconds.

Example 2.9. Injecting Transport Options into a Discovered Transport

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

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

2.2.2. Discovery Agents

Abstract

A discovery agent is a mechanism that advertises available brokers to clients and other brokers.

2.2.2.1. Introduction to Discovery Agents

What is a discovery agent?

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.
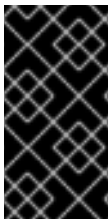
Discovery mechanisms

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 2.10, "Enabling a Discovery Agent on a Broker".

Example 2.10. 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**.

> **IMPORTANT**
>
> 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 undiscoverable.

Discovery agent types

Red Hat JBoss A-MQ currently supports the following discovery agents:

- Fuse Fabric Discovery Agent

- Static Discovery Agent

- Multicast Discovery Agent

- Zeroconf Discovery Agent

2.2.2.2. Fuse Fabric Discovery Agent

Abstract

The Fuse Fabric discovery agent uses Fuse Fabric to discovery brokers that are deployed into a fabric.

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 2.11, "Fuse Fabric Discovery Agent URI Format".

> **Example 2.11. 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 the management console. For information on using the management console see "Management Console User Guide".

You can also use the console to deploy brokers into a fabric. See chapter "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 2.12, "Client Connection URL using Fuse Fabric Discovery" .

> **Example 2.12. Client Connection URL using Fuse Fabric Discovery**
>
> discovery:(fabric://nwBrokers)

A client using the URL in Example 2.12, "Client Connection URL using Fuse Fabric Discovery" will discover all the brokers in the **nwBrokers** broker group and generate a list of brokers to which it can connect.

2.2.2.3. Static Discovery Agent

Abstract

The static discovery agent uses an explicit list of broker URLs to specify the available brokers.

Overview

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

Using the agent

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

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

The static discovery agent URI conforms to the syntax in Example 2.13, "Static Discovery Agent URI Format".

> **Example 2.13. Static Discovery Agent URI Format**
>
> static://(*URI1*,*URI2*,*URI3*,...)

Example

Example 2.14, "Discovery URI using the Static Discovery Agent" shows a discovery URI that configures a client to use the static discovery agent to connect to one member of a broker pair.

> **Example 2.14. Discovery URI using the Static Discovery Agent**
>
> discovery:(static://(tcp://localhost:61716,tcp://localhost:61816))

2.2.2.4. Multicast Discovery Agent

Abstract

The multicast discovery agent uses the IP multicast protocol to find any message brokers currently active on the local network.

Overview

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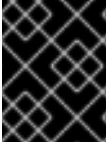
> **IMPORTANT**
>
> Your local network (LAN) must be configured appropriately for the IP/multicast protocol to work.

URI

The multicast discovery agent URI conforms to the syntax in Example 2.15, "Multicast Discovery Agent URI Format".

Example 2.15. 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 2.16, "Enabling a Multicast Discovery Agent on a Broker" .

Example 2.16. Enabling a Multicast Discovery Agent on a Broker

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

The broker configured in Example 2.16, "Enabling a Multicast Discovery Agent on a Broker" 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 2.17, "Client Connection URL using Multicast Discovery" .

Example 2.17. Client Connection URL using Multicast Discovery

discovery:(multicast://default)

A client using the URI in Example 2.17, "Client Connection URL using Multicast Discovery" will discover all the brokers advertised in the **default** multicast group and generate a list of brokers to which it can connect.

2.2.2.5. Zeroconf Discovery Agent

Abstract

The zeroconf discovery agent uses an open source implementation of Apple's Bonjour networking technology to find any brokers currently active on the local network.

Overview

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

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

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

URI

The zeroconf discovery agent URI conforms to the syntax in Example 2.18, "Zeroconf Discovery Agent URI Format".

Example 2.18. 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 2.19, "Enabling a Multicast Discovery Agent on a Broker" .

Example 2.19. 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 2.19, "Enabling a Multicast Discovery Agent on a Broker" is discoverable as part of the multicast group **NEGroup**.

Configuring a client

To use the agent a client must be configured to connect to a broker using a discovery protocol that uses a zeroconf agent URI as shown in Example 2.20, "Client Connection URL using Zeroconf Discovery" .

> **Example 2.20. Client Connection URL using Zeroconf Discovery**
>
> > discovery:(zeroconf://NEGroup)

A client using the URL in Example 2.20, "Client Connection URL using Zeroconf Discovery" will discover all the brokers advertised in the **NEGroup** multicast group and generate a list of brokers to which it can connect.

CHAPTER 3. MASTER/SLAVE

Abstract

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

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

Red Hat JBoss A-MQ supports two master/slave broker configurations:

- Shared file system—the master and the slaves use a common persistence store that is located on a shared file system

- Shared JDBC database—the masters and the slaves use a common JDBC persistence store

3.1. SHARED FILE SYSTEM MASTER/SLAVE

Overview

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

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

Supported network file systems

The following network file systems (and only these file systems) are supported by AMQ:

- NFSv4

- GFS2

- CIFS/SMB (Windows only)

Recommended NFSv4 client mount options

The goal is to set mount options that provide optimal support for both broker failover and data persistence. For broker failover, you want errors to propagate up the broker quickly. For data persistence, you want to resend failed requests many times. The trick is to find settings that optimally balance both fault tolerant messaging features.

The following mount options were used in all NFS locking mechanism tests. The tests were run on Red Hat Enterprise Linux 7.x machines in Red Hat OpenStack Platform. The broker was configured to use the KahaDB store with **lockKeepAlivePeriod=2000** (for details, see the section called "File locking

requirements"). In these tests, the broker detected lost access to the data store and initiated shutdown within 12 seconds. You may need to adjust these settings depending on your particular setup.

- **soft**—Disables continuous retransmission attempts by the client when the NFS server does not respond to a request. Instead, an NFS request fails after **retrans** transmissions have been sent, causing the NFS client to return an error to the calling client, and thus the broker. This option is key for enabling the **timeo** and **retrans** options.

- **timeo=**_20_—The time, in deciseconds, the NFS client waits for a response from the NFS server, before it sends another request. The default is **600** (60 seconds).

- **retrans=**_2_—Specifies the number of times the NFS client attempts to retransmit a failed request to the NFS server. The default is **3**. The client waits a **timeo** timeout period between each **retrans** attempt.
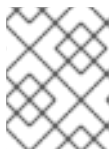
> **NOTE**
>
> After each retransmission, the timeout period is incremented by **timeo**, up to the maximum allowed (**600**).

- **lookupcache=**_none_—Specifies how the kernel manages its cache of directory entries for the mount point. **none** forces the client to revalidate all cache entries before they are used. This enables the Master broker to immediately detect any change made to the lock file, and it prevents the lock checking mechanism from returning incorrect validity information.

  The default is **all**, which means the client assumes that all cache directory entries are valid until their parent directory's cached attributes expire.

- **sync**—Any system call that writes data to files on the mount point causes the data to be flushed to the NFS server before the system call returns control to user space. This option provides greater data cache coherence.

- **intr**—Allows signals to interrupt file operations on the mount point. System calls return **EINTR** when an in-progress NFS operation is interrupted by a signal.

- **proto=**_tcp_—Specifies the protocol the NFS client uses to transmit requests to the NFS server.

For more information on NFS mount point options, see http://linux.die.net/man/5/nfs.

File locking requirements

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

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

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

NFSv4, on the other hand, *is* compatible with this master/slave configuration, as its design includes timeouts for locks. When an NFSv4 client holding a lock terminates abnormally, NFSv4 automatically releases the lock after the specified timeout (see http://tools.ietf.org/html/rfc5661 for details), allowing another NFSv4 client to grab it.

It is possible for a slave to grab the lock from the master without the master's knowledge when NFSv4 crashes. This is so because the master broker does not automatically check whether it still has the lock, giving a slave the chance to grab it when the NFSv4 specified timeout elapses.

You can avoid this scenario by using the persistence adapter's **lockKeepAlivePeriod** attribute. Setting the **lockKeepAlivePeriod** attribute instructs the master to check, at intervals of the specified milliseconds, whether it still holds the lock (lock is valid) and that the lock file still exists. If the master discovers that the lock is invalid, it tries to regain the lock. If it fails or the lock file no longer exists, the master enters Slave mode, allowing another slave to try to get the lock and become master.

In attempting to get the lock, the slave checks every **lockAcquireSleepInterval** (milliseconds) whether another broker holds the lock. If not, the slave locks the file and waits one **lockKeepAlivePeriod** before entering Master mode. If the lock file does not exist, the slave recreates it and then tries to lock it, following the same procedure it would if the lock file existed.

To enable this lock checking mechanism, add the **lockKeepAlivePeriod** attribute to the **persistence Adaptor** element in the broker configuration. For example, like this:

```
<kahaDB directory="/sharedFileSystem/sharedBrokerData" lockKeepAlivePeriod="2000">
   <locker>
      <shared-file-locker lockAcquireSleepInterval="10000" />
   </locker>
</kahaDB>
```

which instructs the master broker to check at five second intervals whether the lock is still valid and that the lock file exists. Example 3.1, "Shared File System Broker Configuration" shows how to set the **lockAcquireSleepInterval** attribute.

Initial state

Figure 3.1, "Shared File System Initial State" shows the initial state of a shared file system master/slave group. When all of the brokers are started, one of them grabs the exclusive lock on the broker data store and becomes the master. All of the other brokers remain slaves and pause while waiting for the exclusive lock to be freed up. Only the master starts its transport connectors, so all of the clients connect to it.

Figure 3.1. Shared File System Initial State
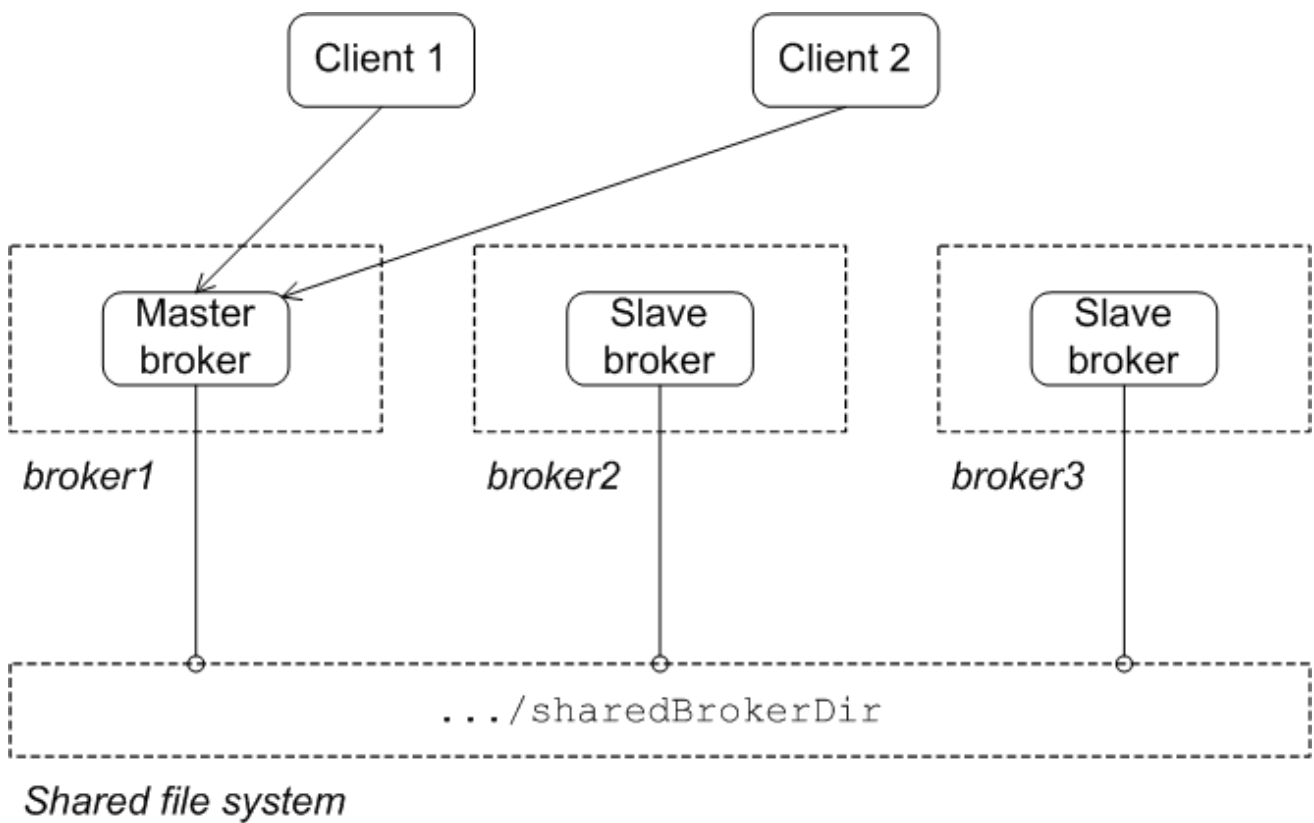
State after failure of the master

Figure 3.2, "Shared File System after Master Failure" shows the state of the master/slave group after the original master has shut down or failed. As soon as the master gives up the lock (or after a suitable timeout, if the master crashes), the lock on the data store frees up and another broker grabs the lock and gets promoted to master.

Figure 3.2. Shared File System after Master Failure
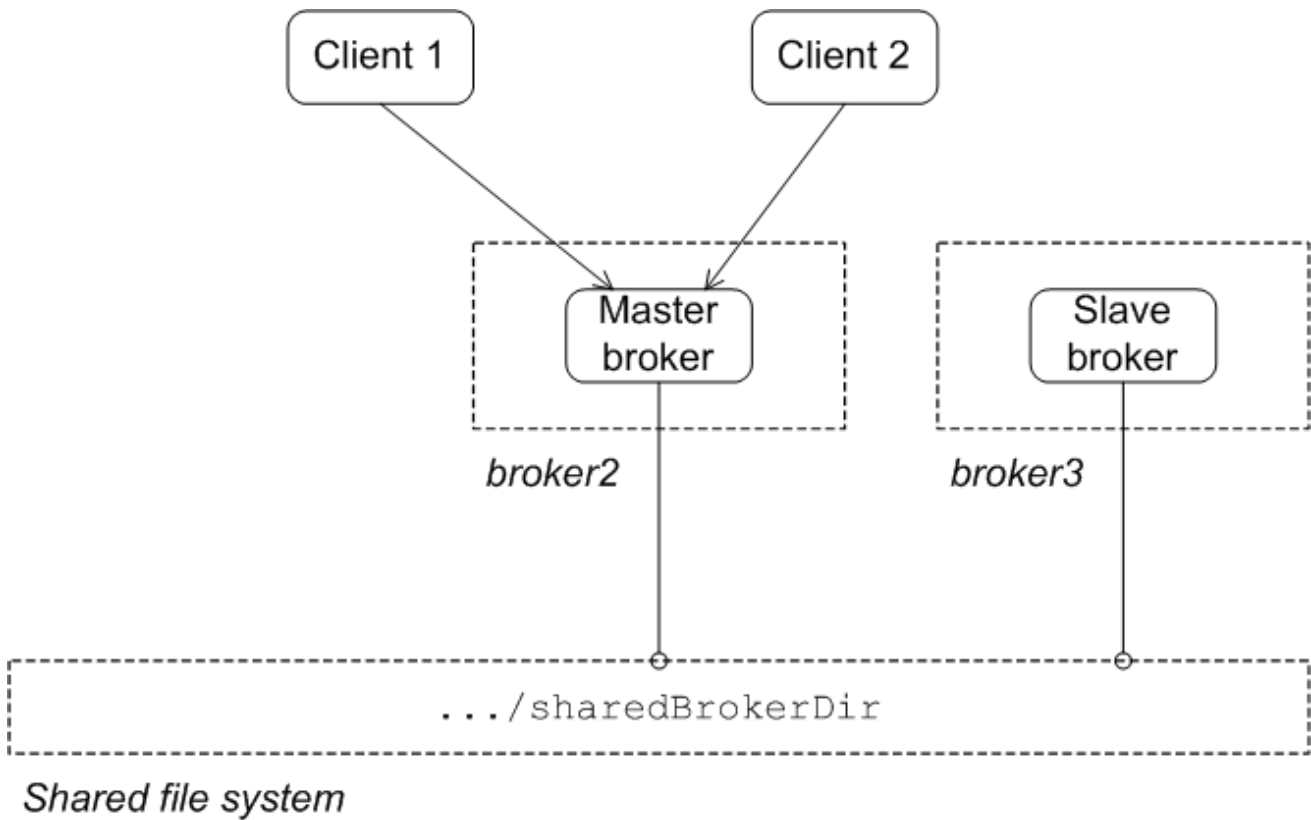
After the clients lose their connection to the original master, they automatically try all of the other brokers listed in the failover URL. This enables them to find and connect to the new master.

Configuring the brokers

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

Example 3.1, "Shared File System Broker Configuration" shows the broker configuration for a shared file system master/slave group that shares a data store located at **/sharedFileSystem/sharedBrokerData** and uses the KahaDB persistence store.

Example 3.1. Shared File System Broker Configuration

```
<broker ... >
  ...
  <persistenceAdapter>
    <kahaDB directory="/sharedFileSystem/sharedBrokerData" lockKeepAlivePeriod="2000">
       <locker>
          <shared-file-locker lockAcquireSleepInterval="10000" />
       </locker>
    </kahaDB>
  </persistenceAdapter>
  ...
</broker>
```

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

Configuring the clients

Clients of shared file system master/slave group must be configured with a failover URL that lists the URLs for all of the brokers in the group. Example 3.2, "Client URL for a Shared File System Master/Slave Group" shows the client failover URL for a group that consists of three brokers: **broker1**, **broker2**, and **broker3**.

> **Example 3.2. Client URL for a Shared File System Master/Slave Group**
>
> > failover:(tcp://broker1:61616,tcp://broker2:61616,tcp://broker3:61616)

For more information about using the failover protocol see Section 2.1.1, "Static Failover".

Reintroducing a failed node

You can restart the failed master at any time and it will rejoin the cluster. It will rejoin as a slave broker because one of the other brokers already owns the exclusive lock on the data store, as shown in Figure 3.3, "Shared File System after Master Restart" .

Figure 3.3. Shared File System after Master Restart
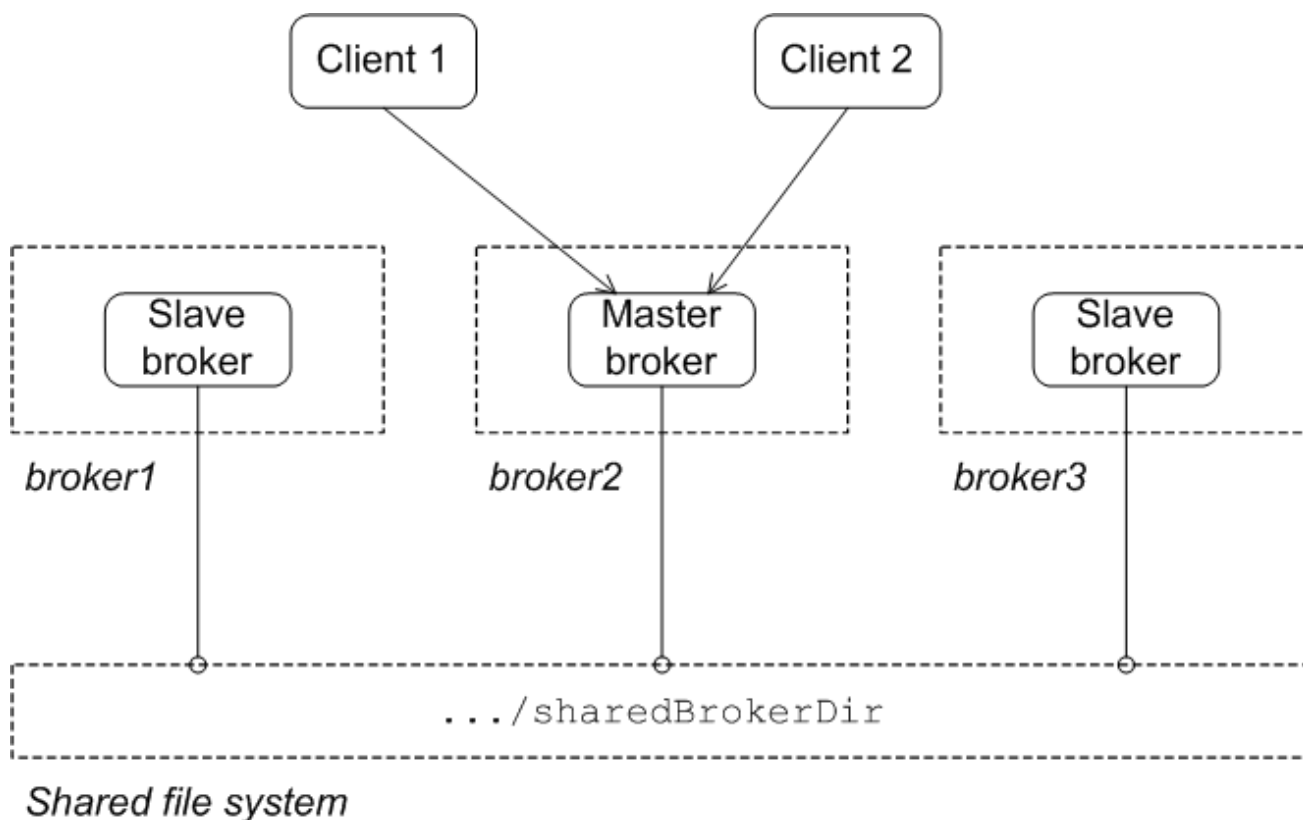
3.2. SHARED JDBC MASTER/SLAVE

Overview

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

The disadvantages of this configuration are:

- The shared database is a single point of failure. This disadvantage can be mitigated by using a database with built in high availability(HA) functionality. The database will handle replication and fail over of the data store.

- You cannot enable high speed journaling. This has a significant impact on performance.

Initial state

Figure 3.4, "JDBC Master/Slave Initial State" shows the initial state of a JDBC master/slave group. When all of the brokers are started, one of them grabs the mutex lock on the database table and becomes the master. All of the other brokers become slaves and pause while waiting for the lock to be freed up. Only the master starts its transport connectors, so all of the clients connect to it.

Figure 3.4. JDBC Master/Slave Initial State
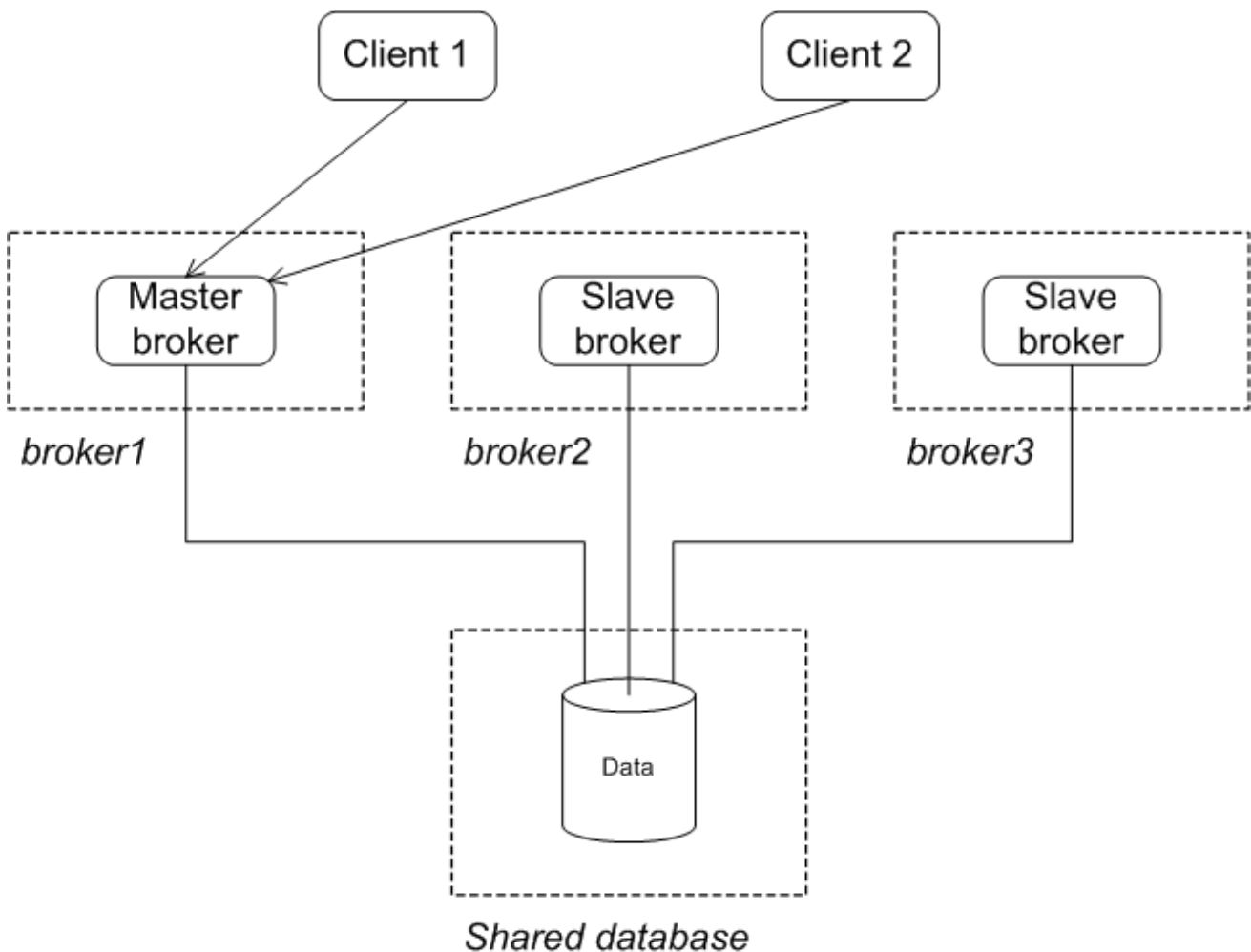
After failure of the master

Figure 3.5, "JDBC Master/Slave after Master Failure" shows the state of the group after the original master has shut down or failed. As soon as the master gives up the lock (or after a suitable timeout, if the master crashes), the lock on the database table frees up and another broker grabs the lock and gets promoted to master.

Figure 3.5. JDBC Master/Slave after Master Failure
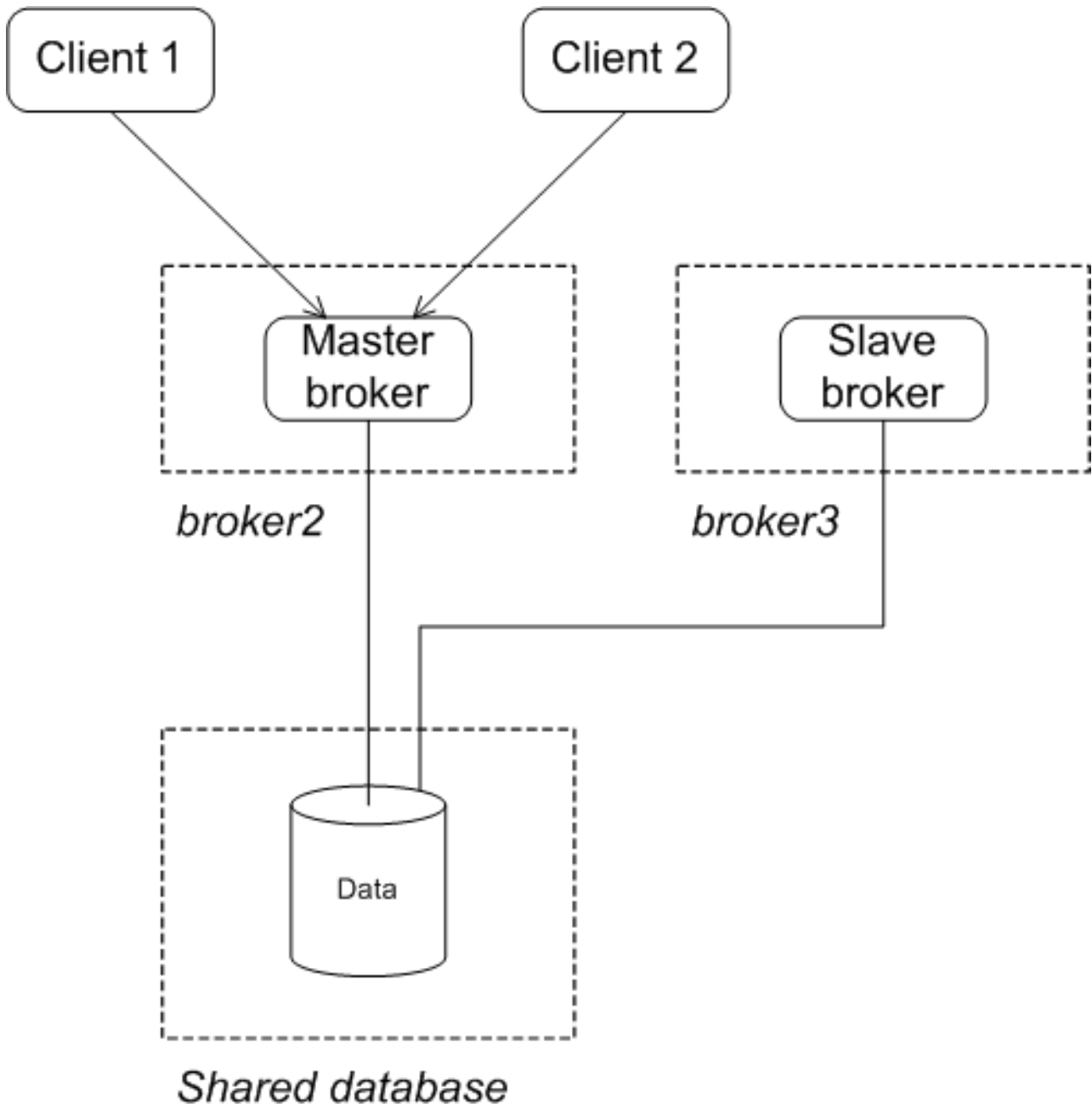
After the clients lose their connection to the original master, they automatically try all of the other brokers listed in the failover URL. This enables them to find and connect to the new master.

Configuring the brokers

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

There is one important requirement when configuring the JDBC persistence adapter for use in a shared database master/slave cluster. You **must** use the direct JDBC persistence adapter. This is because the journal used by the journaled JDBC persistence adapter is not replicated and batch updates are used to sync with the JDBC store. Therefore it is not possible to guarantee that the latest updates are on the shared JDBC store.

Example 3.3, "JDBC Master/Slave Broker Configuration" shows the configuration used be a master/slave group that stores the shared broker data in an Oracle database.

Example 3.3. 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.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>
```

NOTE

Share the scheduled messages between the brokers and adjust the schedule store directory along with the **schedulerDirectory** as shown below:

```
<broker xmlns="http://activemq.apache.org/schema/core" schedulerSupport="true"
schedulerDirectory="{activemq.data}/broker/scheduler"/>
```

Configuring the clients

Clients of shared JDBC master/slave group must be configured with a failover URL that lists the URLs for all of the brokers in the group. Example 3.4, "Client URL for a Shared JDBC Master/Slave Group" shows the client failover URL for a group that consists of three brokers: **broker1**, **broker2**, and **broker3**.

**Example 3.4. Client URL for a Shared JDBC Master/Slave Group**

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

For more information about using the failover protocol see Section 2.1.1, "Static Failover".

Reintroducing a failed node

You can restart the failed node at any time and it will rejoin the group. It will rejoin the group as a slave because one of the other brokers already owns the mutex lock on the database table, as shown in Figure 3.6, "JDBC Master/Slave after Master Restart".

Figure 3.6. JDBC Master/Slave after Master Restart
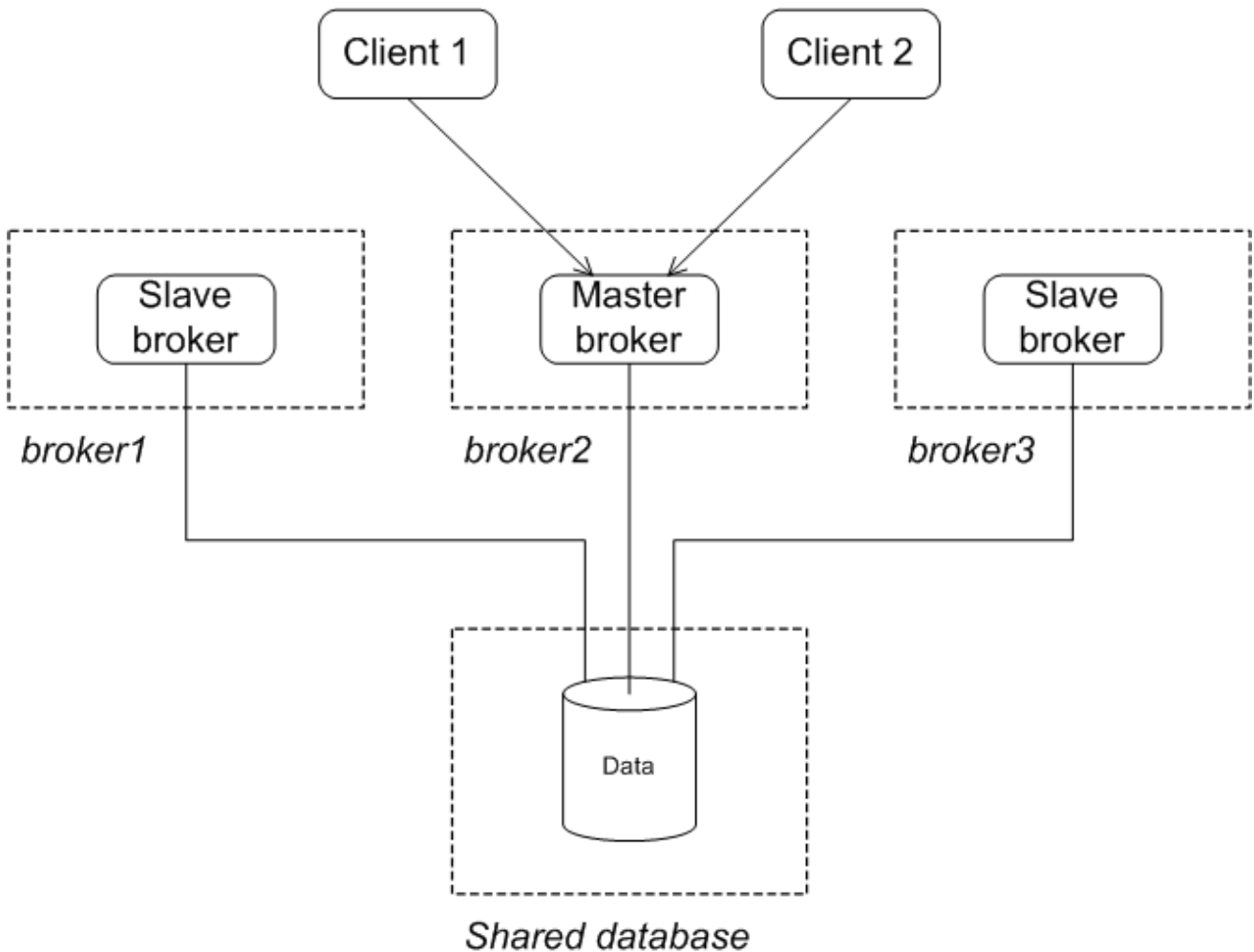
CHAPTER 4. MASTER/SLAVE AND BROKER NETWORKS

Abstract

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

OVERVIEW

Master/slave groups and broker networks represent different levels of organization. A network of brokers provides a symmetrical group of brokers that share information among all of the members in the group. They are useful for distributing the message processing load among many brokers.

Master/slave groups are asymmetrical> Only one member of the group is active at a time. They are useful for providing fault tolerance when data loss is unacceptable.

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

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

- Network connectors to a master/slave group use a special protocol.

- A broker cannot open a network connection to another member of its master/slave group.

CONFIGURING THE CONNECTION TO A MASTER/SLAVE GROUP

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

- Open a connection to the master broker without connecting to the slave brokers.

- Connect to the new master in the case of a failure.

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

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

Example 4.1, "Network Connector to a Master/Slave Group" shows a network connector configured to link to a master/slave group.

Example 4.1. Network Connector to a Master/Slave Group

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

HOST PAIR WITH MASTER/SLAVE GROUPS

In order to scale up to a large fault tolerant broker network, it is a good idea to adopt a simple building block as the basis for the network. An effective building block for this purpose is the host pair arrangement shown in Figure 4.1, "Master/Slave Groups on Two Host Machines" .

Figure 4.1. Master/Slave Groups on Two Host Machines
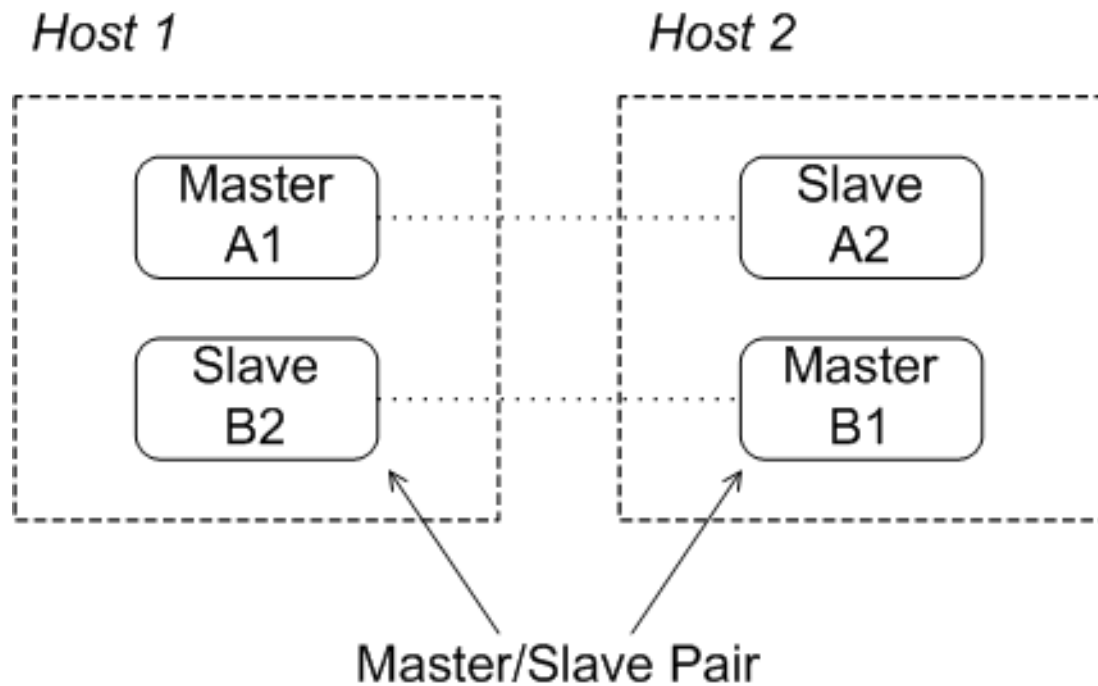
The host pair arrangement consists of two master/slave groups distributed between two host machines. Under normal operating conditions, one master broker is active on each of the two host machines. If one of the machines should fail for some reason, the slave on the other machine takes over, so that you end up with two active brokers on the healthy machine.

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

NETWORK OF MULTIPLE HOST PAIRS

You can easily scale up to a large fault tolerant broker network by adding host pairs, as shown in Figure 4.2, "Broker Network Consisting of Host Pairs" .

Figure 4.2. Broker Network Consisting of Host Pairs
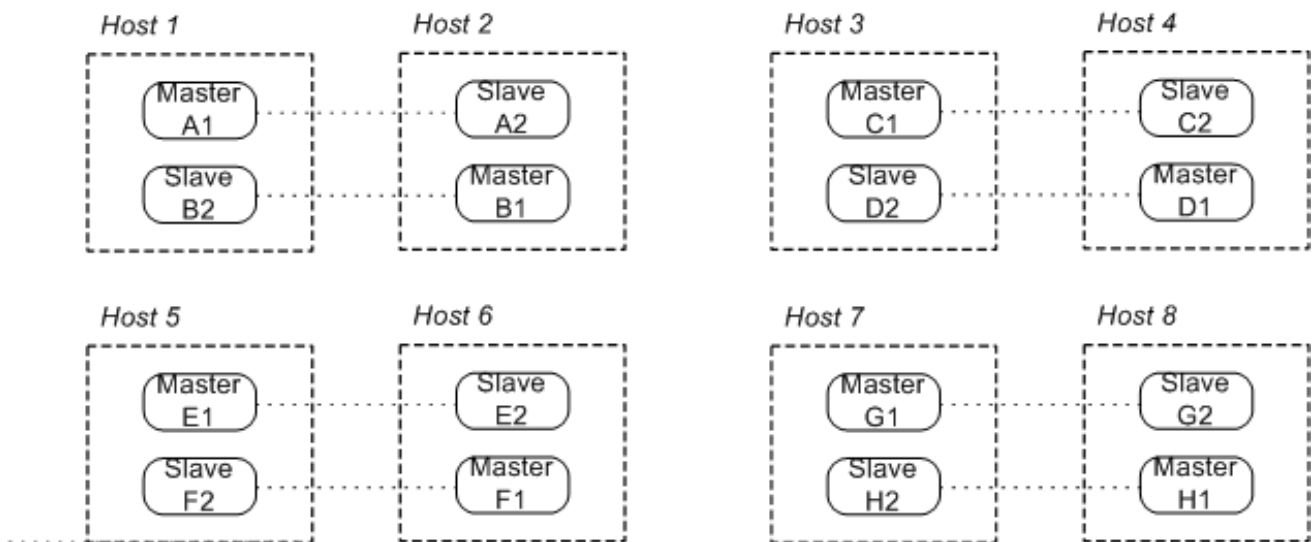
The preceding network consists of eight master/slave groups distributed over eight host machines. As before, you should open network connectors only to brokers outside the current master/slave group. For example, **brokerA1** can connect to at most the following brokers: **brokerB***, **brokerC***, **brokerD***, **brokerE***, **brokerF***, **brokerG***, and **brokerH***.

MORE INFORMATION

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

INDEX

B

broker networks

master/slave, Configuring the connection to a master/slave group

broker properties

rebalanceClusterClients, Broker-side configuration

updateClusterClients, Broker-side configuration

updateClusterClientsOnRemove, Broker-side configuration

updateClusterFilter, Broker-side configuration

D

discovery agent

Fuse Fabric, Fuse Fabric Discovery Agent

multicast, Multicast Discovery Agent

static, Static Discovery Agent

zeroconf, Zeroconf Discovery Agent

discovery protocol

F

P

persistenceAdapter, Configuring the brokers, Configuring the brokers

S

shared file system master/slave

    advantages, Overview

    broker configuration, Configuring the brokers, Configuring the brokers

    client configuration, Configuring the clients

    disadvantages, Overview

    incompatible SANs, File locking requirements

    initial state, Initial state

    master failure, State after failure of the master

    NFSv3, File locking requirements

    NFSv4, File locking requirements

    OCFS2, File locking requirements

    recovery strategies, State after failure of the master

    reintroducing a node, Reintroducing a failed node

shared JDBC master/slave

    advantages, Overview

    client configuration, Configuring the clients

    disadvantages, Overview

    initial state, Initial state

    master failure, After failure of the master

    recovery strategies, After failure of the master

    reintroducing a node, Reintroducing a failed node

static discovery agent

    URI, Using the agent

static failover, Static Failover

static://, Using the agent

T

transportConnector

    discoveryUri, Configuring a broker, Configuring a broker

Z

zeroconf discovery agent

broker configuration, Configuring a broker

URI, URI

zeroconf://, URI