



# **Red Hat AMQ 7.2**

## **Using AMQ Interconnect**

For Use with AMQ Interconnect 1.2



# Red Hat AMQ 7.2 Using AMQ Interconnect

---

For Use with AMQ Interconnect 1.2

## Legal Notice

Copyright © 2018 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## Abstract

This guide describes how to install, configure, and manage AMQ Interconnect to build a large-scale messaging network.

## Table of Contents

<b>CHAPTER 1. OVERVIEW</b>	<b>5</b>
1.1. KEY FEATURES	5
1.2. SUPPORTED STANDARDS AND PROTOCOLS	5
1.3. SUPPORTED CONFIGURATIONS	5
1.4. THEORY OF OPERATION	5
1.4.1. Overview	5
1.4.2. Connections	6
1.4.2.1. Listener	6
1.4.2.2. Connector	6
1.4.3. Addresses	7
1.4.3.1. Mobile Addresses	7
1.4.3.1.1. Discovered Mobile Addresses	8
1.4.3.1.2. Configured Mobile Addresses	8
1.4.3.2. Link Route Addresses	9
1.4.4. Message Routing	9
1.4.4.1. Routing Patterns	9
1.4.4.2. Routing Mechanisms	9
1.4.4.2.1. Message Routed	10
1.4.4.2.2. Link Routed	10
1.4.4.3. Message Settlement	10
1.4.5. Security	11
1.5. DOCUMENT CONVENTIONS	11
<b>CHAPTER 2. INSTALLATION</b>	<b>12</b>
<b>CHAPTER 3. UPGRADING AMQ INTERCONNECT</b>	<b>13</b>
<b>CHAPTER 4. GETTING STARTED</b>	<b>14</b>
4.1. STARTING THE ROUTER	14
4.2. ROUTING MESSAGES IN A PEER-TO-PEER CONFIGURATION	15
4.2.1. Starting the Receiver Client	16
4.2.2. Sending Messages	16
<b>CHAPTER 5. CONFIGURATION</b>	<b>18</b>
5.1. ACCESSING THE ROUTER CONFIGURATION FILE	18
5.2. HOW THE ROUTER CONFIGURATION FILE IS STRUCTURED	18
5.3. METHODS FOR USING PATTERN MATCHING AND WILDCARDS	18
5.3.1. Pattern Matching for Addresses	18
5.3.2. Pattern Matching for Vhost Policy Hostnames	19
5.4. CHANGING A ROUTER'S CONFIGURATION	20
5.4.1. Making a Permanent Change to the Router's Configuration	20
5.4.2. Changing the Configuration for a Running Router	21
5.5. DEFAULT CONFIGURATION SETTINGS	21
5.6. SETTING ESSENTIAL CONFIGURATION PROPERTIES	22
<b>CHAPTER 6. NETWORK CONNECTIONS</b>	<b>24</b>
6.1. LISTENING FOR INCOMING CONNECTIONS	24
6.2. ADDING OUTGOING CONNECTIONS	24
<b>CHAPTER 7. SECURITY</b>	<b>26</b>
7.1. AUTHENTICATING REMOTE PEERS	26
7.1.1. Setting Up SSL/TLS for Encryption and Authentication	26
7.1.2. Setting Up SASL for Authentication and Payload Encryption	28

7.1.3. Securing Incoming Connections	29
7.1.3.1. Adding SSL/TLS Encryption to an Incoming Connection	29
7.1.3.2. Adding SASL Authentication to an Incoming Connection	30
7.1.3.3. Adding SSL/TLS Client Authentication to an Incoming Connection	30
7.1.3.4. Adding SASL Payload Encryption to an Incoming Connection	31
7.1.4. Securing Outgoing Connections	31
7.1.4.1. Adding SSL/TLS Client Authentication to an Outgoing Connection	31
7.1.4.2. Adding SASL Authentication to an Outgoing Connection	32
7.2. INTEGRATING WITH KERBEROS	32
7.3. AUTHORIZING ACCESS TO MESSAGING RESOURCES	33
7.3.1. How AMQ Interconnect Enforces Connection and Resource Limits	34
7.3.2. Setting Global Connection Limits	34
7.3.3. Setting Connection and Resource Limits for Messaging Endpoints	34
7.3.3.1. Enabling Vhost Policies	35
7.3.3.2. Configuring Vhost Policies in the Router Configuration File	35
7.3.3.3. Configuring Vhost Policies as JSON Files	38
7.3.3.4. Methods for Specifying Vhost Policy Source and Target Addresses	39
7.3.3.5. Vhost Policy Examples	40
<b>CHAPTER 8. ROUTING</b>	<b>43</b>
8.1. COMPARISON OF MESSAGE ROUTING AND LINK ROUTING	44
8.1.1. When to Use Message Routing	44
8.1.2. When to Use Link Routing	44
8.2. CONFIGURING MESSAGE ROUTING	45
8.2.1. Addresses	45
8.2.2. Routing Patterns	46
8.2.3. Message Settlement	47
8.2.4. Routing Messages Between Clients	48
8.2.5. Routing Messages Through a Broker Queue	49
8.2.5.1. Configuring Waypoint Addresses	51
8.2.5.2. Connecting a Router to the Broker	52
8.2.6. Example: Routing Messages Through Broker Queues	54
8.2.6.1. Router Configuration	54
8.2.6.2. How the Messages are Routed	55
8.3. CONFIGURING LINK ROUTING	56
8.3.1. Link Route Addresses	57
8.3.2. Link Route Routing Patterns	57
8.3.3. Link Route Flow Control	57
8.3.4. Creating a Link Route	57
8.3.5. Example: Using a Link Route to Provide Client Isolation	60
8.3.5.1. Router Configuration	61
8.3.5.2. How the Client Receives Messages	62
<b>CHAPTER 9. LOGGING</b>	<b>63</b>
9.1. LOGGING MODULES	63
9.1.1. The DEFAULT Logging Module	63
9.1.2. The ROUTER Logging Module	63
9.1.3. The ROUTER_CORE Logging Module	63
9.1.4. The ROUTER_HELLO Logging Module	64
9.1.5. The ROUTER_LS Logging Module	64
9.1.6. The ROUTER_MA Logging Module	65
9.1.7. The MESSAGE Logging Module	66
9.1.8. The SERVER Logging Module	67

9.1.9. The AGENT Logging Module	67
9.1.10. The CONTAINER Logging Module	68
9.1.11. The ERROR Logging Module	68
9.1.12. The POLICY Logging Module	69
9.2. CONFIGURING LOGGING	69
9.3. VIEWING LOG ENTRIES	71
9.3.1. Viewing Log Entries on the Console	71
9.3.2. Viewing Log Entries on the CLI	71
<b>CHAPTER 10. MANAGEMENT</b>	<b>72</b>
10.1. USING AMQ CONSOLE	72
10.2. MONITORING AMQ INTERCONNECT USING QDSTAT	72
10.2.1. Syntax for Using qdstat	72
10.2.2. Viewing General Statistics for a Router	72
10.2.3. Viewing a List of Connections to a Router	73
10.2.4. Viewing AMQP Links Attached to a Router	74
10.2.5. Viewing Known Routers on a Network	76
10.2.6. Viewing Addresses Known to a Router	78
10.2.7. Viewing a Router's Autolinks	79
10.2.8. Viewing the Status of a Router's Link Routes	80
10.2.9. Viewing Memory Consumption Information	80
10.3. MANAGING AMQ INTERCONNECT USING QDMANAGE	81
10.3.1. Syntax for Using qdmanage	81
10.3.2. Managing Network Connections	82
10.3.2.1. Managing Listeners	82
10.3.2.2. Managing Connectors	84
10.3.3. Managing Security	85
10.3.3.1. Managing SSL/TLS Encryption and Authentication	85
10.3.3.2. Managing SASL Encryption and Authentication	87
10.3.4. Managing Routing	88
10.3.4.1. Managing Message Routing	88
10.3.4.2. Managing Link Routing	90
10.3.5. Managing Logging	91
<b>CHAPTER 11. RELIABILITY</b>	<b>93</b>
11.1. PATH REDUNDANCY	93
11.2. PATH REDUNDANCY AND TEMPORAL DECOUPLING	97
11.3. SHARDED QUEUE	105
<b>APPENDIX A. USING CYRUS SASL TO PROVIDE AUTHENTICATION</b>	<b>109</b>
A.1. GENERATING A SASL DATABASE	109
A.2. VIEWING USERS IN A SASL DATABASE	109
A.3. CONFIGURING A SASL DATABASE	109
<b>APPENDIX B. USING YOUR SUBSCRIPTION</b>	<b>111</b>
Accessing your account	111
Activating a subscription	111
Downloading zip and tar files	111
Registering your system for packages	111





# CHAPTER 1. OVERVIEW

AMQ Interconnect is a lightweight AMQP message router for building scalable, available, and performant messaging networks.

AMQ Interconnect is based on Dispatch Router from the [Apache Qpid™](#) project.

## 1.1. KEY FEATURES

- Connects clients and brokers into an internet-scale messaging network with uniform addressing
- Supports high-performance direct messaging
- Uses redundant network paths to route around failures
- Streamlines the management of large deployments

## 1.2. SUPPORTED STANDARDS AND PROTOCOLS

AMQ Interconnect supports the following industry-recognized standards and network protocols:

- Version 1.0 of the [Advanced Message Queueing Protocol](#) (AMQP)
- Modern [TCP](#) with [IPv6](#)



### NOTE

The details of distributed transactions (XA) within AMQP are not provided in the 1.0 version of the specification. AMQ Interconnect does not support XA transactions.

## 1.3. SUPPORTED CONFIGURATIONS

AMQ Interconnect is supported on Red Hat Enterprise Linux 6 and 7. See [Red Hat AMQ 7 Supported Configurations](#) for more information.

## 1.4. THEORY OF OPERATION

This section introduces some key concepts about AMQ Interconnect

### 1.4.1. Overview

AMQ Interconnect is an *application layer* program running as a normal user program or as a daemon.

The router accepts AMQP connections from clients and creates AMQP connections to brokers or AMQP-based services. The router classifies incoming AMQP messages and routes the messages between message producers and message consumers.

The router is meant to be deployed in topologies of multiple routers, preferably with redundant paths. It uses link-state routing protocols and algorithms similar to OSPF or IS-IS from the networking world to calculate the best path from every message source to every message destination and to recover quickly from failures. The router relies on redundant network paths to provide continued connectivity in the face of system or network failure.

A messaging client can make a single AMQP connection into a messaging bus built with routers and, over that connection, exchange messages with one or more message brokers connected to any router in the network. At the same time the client can exchange messages directly with other endpoints without involving a broker at all.

## 1.4.2. Connections

AMQ Interconnect connects clients, servers, AMQP services, and other routers through network connections.

### 1.4.2.1. Listener

The router provides *listeners* that accept client connections. A client connecting to a router listener uses the same methods that it would use to connect to a broker. From the client's perspective the router connection and link establishment are identical to broker connection and link establishment.

Several types of listeners are defined by their role.

Role	Description
<b>normal</b>	The connection is used for AMQP clients using normal message delivery.
<b>inter-router</b>	The connection is assumed to be to another router in the network. Inter-router discovery and routing protocols can only be used over inter-router connections.
<b>route-container</b>	The connection is a broker or other resource that holds known addresses. The router will use this connection to create links as necessary. The addresses are available for routing only after the remote resource has created a connection.

### 1.4.2.2. Connector

The router can also be configured to create outbound connections to messaging brokers or other AMQP entities using *connectors*. A connector is defined with the network address of the broker and the name or names of the resources that are available in that broker. When a router connects to a broker through a connector it uses the same methods a normal messaging client would use when connecting to the broker.

Several types of connectors are defined by their role.

Role	Description
<b>normal</b>	The connection is used for AMQP clients using normal message delivery. On this connector the router will initiate the connection but it will never create any links. Links are to be created by the peer that accepts the connection.
<b>inter-router</b>	The connection is assumed to be to another router in the network. Inter-router discovery and routing protocols can only be used over inter-router connections.
<b>route-container</b>	The connection is to a broker or other resource that holds known addresses. The router will use this connection to create links as necessary. The addresses are available for routing only after the router has created a connection to the remote resource.

### 1.4.3. Addresses

AMQP addresses are used to control the flow of messages across a network of routers. Addresses are used in a number of different places in the AMQP 1.0 protocol. They can be used in a specific message in the *to* and *reply-to* fields of a message's properties. They are also used during the creation of links in the *address* field of a *source* or a *target*.



#### NOTE

Addresses in this discussion refer to AMQP protocol addresses and not to TCP/IP network addresses. TCP/IP network addresses are used by messaging clients, brokers, and routers to create AMQP connections. AMQP protocol addresses are the names of source and destination endpoints for messages within the messaging network.

Addresses designate various kinds of entities in a messaging network:

- Endpoint processes that consume data or offer a service
- Topics that match multiple consumers to multiple producers
- Entities within a messaging broker:
  - Queues
  - Durable Topics
  - Exchanges

The syntax of an AMQP address is opaque as far as the router network is concerned. A syntactical structure may be used by the administrator who creates addresses but the router treats them as opaque strings.

The router maintains several classes of address based on how the address is configured or discovered.

Address Type	Description
Mobile	The address is a rendezvous point between senders and receivers. The router aggregates and serializes messages from senders and distributes messages to receivers.
Link route	The address defines a private messaging path between a sender and a receiver. The router simply passes messages between the end points.

#### 1.4.3.1. Mobile Addresses

Routers consider addresses to be mobile such that any users of an address may be directly connected to any router in a network and may move around the topology. In cases where messages are broadcast to or balanced across multiple consumers, the address users may be connected to multiple routers in the network.

Mobile addresses are rendezvous points for senders and receivers. Messages arrive at the mobile address and are dispatched to their destinations according to the routing defined for the mobile address. The details of these routing patterns are discussed later.

Mobile addresses may be discovered during normal router operation or configured through management settings.

#### 1.4.3.1.1. Discovered Mobile Addresses

Mobile addresses are created when a client creates a link to a source or destination address that is unknown to the router network.

Suppose a service provider wants to offer *my-service* that clients may use. The service provider must open a receiver link with source address *my-service*. The router creates a mobile address *my-service* and propagates the address so that it is known to every router in the network.

Later a client wants to use the service and creates a sending link with target address *my-service*. The router matches the service provider's receiver having source address *my-service* to the client's sender having target address *my-service* and routes messages between the two.

Any number of other clients can create links to the service as well. The clients do not have to know where in the router network the service provider is physically located nor are the clients required to connect to a specific router to use the service. Regardless of how many clients are using the service the service provider needs only a single connection and link into the router network.

Another view of this same scenario is when a client tries to use the service before service provider has connected to the network. In this case the router network creates the mobile address *my-service* as before. However, since the mobile address has only client sender links and no receiver links the router stalls the clients and prevents them from sending any messages. Later, after the service provider connects and creates the receiver link, the router will issue credits to the clients and the messages will begin to flow between the clients and the service.

The service provider can connect, disconnect, and reconnect from a different location without having to change any of the clients or their connections. Imagine having the service running on a laptop. One day the connection is from corporate headquarters and the next day the connection is from some remote location. In this case the service provider's computer will typically have different host IP addresses for each connection. Using the router network the service provider connects to the router network and offers the named service and the clients connect to the router network and consume from the named service. The router network routes messages between the mobile addresses effectively masking host IP addresses of the service provider and the client systems.

#### 1.4.3.1.2. Configured Mobile Addresses

Mobile addresses may be configured using the router *autoLink* object. An address created via an *autoLink* represents a queue, topic, or other service in an external broker. Logically the *autoLink* addresses are treated by the router network as if the broker had connected to the router and offered the services itself.

For each configured mobile address the router will create a single link to the external resource. Messages flow between sender links and receiver links the same regardless if the mobile address was discovered or configured.

Multiple *autoLink* objects may define the same address on multiple brokers. In this case the router network creates a sharded resource split between the brokers. Any client can seamlessly send and receive messages from either broker.

Note that the brokers do not need to be clustered or federated to receive this treatment. The brokers may even be from different vendors or be different versions of the same broker yet still work together to provide a larger service platform.

### 1.4.3.2. Link Route Addresses

Link route addresses may be configured using the router *linkRoute* object. An link route address represents a queue, topic, or other service in an external broker similar to addresses configured by *autoLink* objects. For link route addresses the router propagates a separate link attachment to the broker resource for each incoming client link. The router does not automatically create any links to the broker resource.

Using link route addresses the router network does not participate in aggregated message distribution. The router simply passes message delivery and settlement between the two end points.

### 1.4.4. Message Routing

Addresses have semantics associated with them that are assigned when the address is provisioned or discovered. The semantics of an address control how routers behave when they see the address being used. Address semantics include the following considerations:

- Routing pattern - balanced, closest, multicast
- Routing mechanism - message routed, link routed

#### 1.4.4.1. Routing Patterns

Routing patterns define the paths that a message with a mobile address can take across a network. These routing patterns can be used for both direct routing, in which the router distributes messages between clients without a broker, and indirect routing, in which the router enables clients to exchange messages through a broker.

Pattern	Description
Balanced	An anycast method which allows multiple receivers to use the same address. In this case, messages (or links) are routed to exactly one of the receivers and the network attempts to balance the traffic load across the set of receivers using the same address. This routing delivers messages to receivers based on how quickly they settle the deliveries. Faster receivers get more messages.
Closest	An anycast method in which even if there are more receivers for the same address, every message is sent along the shortest path to reach the destination. This means that only one receiver will get the message. Each message is delivered to the closest receivers in terms of topology cost. If there are multiple receivers with the same lowest cost, deliveries will be spread evenly among those receivers.
Multicast	Having multiple consumers on the same address at the same time, messages are routed such that each consumer receives one copy of the message.

#### 1.4.4.2. Routing Mechanisms

The fact that addresses can be used in different ways suggests that message routing can be accomplished in different ways. Before going into the specifics of the different routing mechanisms, it would be good to first define what is meant by the term *routing*:

In a network built of multiple, interconnected routers 'routing' determines which connection to use to send a message directly to its destination or one step closer to its destination.

Each router serves as the terminus of a collection of incoming and outgoing links. Some of the links are designated for message routing, and others are designated for link routing. In both cases, the links either connect directly to endpoints that produce and consume messages, or they connect to other routers in the network along previously established connections.

#### 1.4.4.2.1. Message Routed

Message routing occurs upon delivery of a message and is done based on the address in the message's *to* field.

When a delivery arrives on an incoming message-routing link, the router extracts the address from the delivered message's *to* field and looks the address up in its routing table. The lookup results in zero or more outgoing links onto which the message shall be resent.

Message routing can also occur without an address in the message's *to* field if the incoming link has a target address. In fact, if the sender uses a link with a target address, the *to* field shall be ignored even if used.

#### 1.4.4.2.2. Link Routed

Link routing occurs when a new link is attached to the router across one of its AMQP connections. It is done based on the *target.address* field of an inbound link and the *source.address* field of an outbound link.

Link routing uses the same routing table that message routing uses. The difference is that the routing occurs during the link-attach operation, and link attaches are propagated along the appropriate path to the destination. What results is a chain of links, connected end-to-end, from source to destination. It is similar to a virtual circuit in a telecom system.

Each router in the chain holds pairs of link termini that are tied together. The router then simply exchanges all deliveries, delivery state changes, and link state changes between the two termini.

The endpoints that use the link chain do not see any difference in behavior between a link chain and a single point-to-point link. All of the features available in the link protocol (flow control, transactional delivery, and so on) are available over a routed link-chain.

#### 1.4.4.3. Message Settlement

Messages may be delivered with varying degrees of reliability.

- At most once
- At least once
- Exactly once

The reliability is negotiated between the client and server during link establishment. The router handles all levels of reliability by treating messages as either *pre-settled* or *unsettled*.

Delivery	Handling
Pre-settled	If the arriving delivery is pre-settled (that is, fire and forget), the incoming delivery shall be settled by the router, and the outgoing deliveries shall also be pre-settled. In other words, the pre-settled nature of the message delivery is propagated across the network to the message's destination.
Unsettled	Unsettled delivery is also propagated across the network. Because unsettled delivery records cannot be discarded, the router tracks the incoming deliveries and keeps the association of the incoming deliveries to the resulting outgoing deliveries. This kept association allows the router to continue to propagate changes in delivery state (settlement and disposition) back and forth along the path which the message traveled.

### 1.4.5. Security

AMQ Interconnect uses the SSL/TLS protocol and related certificates and SASL protocol mechanisms to encrypt and authenticate remote peers. Router listeners act as network servers and router connectors act as network clients. Both connection types may be configured securely with SSL/TLS and SASL.

The router Policy module is an optional authorization mechanism enforcing user connection restrictions and AMQP resource access control.

## 1.5. DOCUMENT CONVENTIONS

In this document, **sudo** is used for any command that requires root privileges. You should always exercise caution when using **sudo**, as any changes can affect the entire system.

For more information about using **sudo**, see [The sudo Command](#).

## CHAPTER 2. INSTALLATION

AMQ Interconnect 1.2 is distributed as a set of RPM packages, which are available through your Red Hat subscription.

### Procedure

1. Ensure your subscription has been activated and your system is registered.  
For more information about using the customer portal to activate your Red Hat subscription and register your system for packages, see [Using Your Subscription](#).
2. Subscribe to the required repositories:

#### Red Hat Enterprise Linux 6

```
$ sudo subscription-manager repos --enable=amq-interconnect-1-for-rhel-6-server-rpms --enable=amq-clients-2-for-rhel-6-server-rpms
```

#### Red Hat Enterprise Linux 7

```
$ sudo subscription-manager repos --enable=amq-interconnect-1-for-rhel-7-server-rpms --enable=amq-clients-2-for-rhel-7-server-rpms
```

3. Use the **yum** command to install the **qpidd-dispatch-router** and **qpidd-dispatch-tools** packages and their dependencies:

```
$ sudo yum install qpidd-dispatch-router qpidd-dispatch-tools
```

4. Use the **which** command to verify that the **qdrouterd** executable is present.

```
$ which qdrouterd
/usr/sbin/qdrouterd
```

The **qdrouterd** executable should be located at **/usr/sbin/qdrouterd**.



## CHAPTER 3. UPGRADING AMQ INTERCONNECT

You should upgrade AMQ Interconnect to the latest version to ensure that you have the latest enhancements and fixes. The upgrade process involves installing the new AMQ Interconnect packages and restarting your routers.

You can use these instructions to upgrade AMQ Interconnect to a new *minor release* or *maintenance release*.

### Minor Release

AMQ Interconnect periodically provides point releases, which are minor updates that include new features, as well as bug and security fixes. If you plan to upgrade from one AMQ Interconnect point release to another, for example, from AMQ Interconnect 1.0 to AMQ Interconnect 1.1, code changes should not be required for applications that do not use private, unsupported, or technical preview components.

### Maintenance Release

AMQ Interconnect also periodically provides maintenance releases that contain bug fixes. Maintenance releases increment the minor release version by the last digit, for example from 1.0.0 to 1.0.1. A maintenance release should not require code changes; however, some maintenance releases might require configuration changes.

### Prerequisites

Before performing an upgrade, you should have reviewed the release notes for the target release to ensure that you understand the new features, enhancements, fixes, and issues. To find the release notes for the target release, see the [Red Hat Customer Portal](#).

### Procedure

1. Upgrade the **qpidd-dispatch-router** and **qpidd-dispatch-tools** packages and their dependencies:

```
$ sudo yum update qpidd-dispatch-router qpidd-dispatch-tools
```

For more information, see [Chapter 2, Installation](#).

2. Restart each router in your router network.  
To avoid disruption, you should restart each router one at a time.

This example restarts a router in Red Hat Enterprise Linux 7:

```
$ systemctl restart qdrouterd.service
```

For more information about starting a router, see [Section 4.1, “Starting the Router”](#).

## CHAPTER 4. GETTING STARTED

Before configuring AMQ Interconnect, you should understand how to start the router, how it is configured by default, and how to use it in a simple peer-to-peer configuration.

### 4.1. STARTING THE ROUTER

#### Procedure

1. To start the router with the default configuration, do one of the following:

To...	Enter this command...
Run the router as a service in Red Hat Enterprise Linux 6	<pre>\$ sudo service qdrouterd start</pre>
Run the router as a service in Red Hat Enterprise Linux 7	<pre>\$ systemctl start qdrouterd.service</pre>
Run the router as a daemon	<pre>\$ qdrouterd -d</pre> To start the router in the foreground, do not use the <b>-d</b> parameter.



#### NOTE

You can specify a different configuration file with which to start the router. For more information, see [Changing a Router's Configuration](#).

The router starts, using the default configuration file stored at **/etc/qpid-dispatch/qdrouterd.conf**.

2. View the log to verify the router status:

```
$ qdstat --log
```

This example shows that the router was correctly installed, is running, and is ready to route traffic between clients:

```
$ qdstat --log
Fri May 20 09:38:03 2017 SERVER (info) Container Name: Router.A 1
Fri May 20 09:38:03 2017 ROUTER (info) Router started in Standalone
mode 2
Fri May 20 09:38:03 2017 ROUTER_CORE (info) Router Core thread
running. 0/Router.A
Fri May 20 09:38:03 2017 ROUTER_CORE (info) In-process subscription
M/$management
Fri May 20 09:38:03 2017 AGENT (info) Activating management agent on
$_management_internal 3
```

```

Fri May 20 09:38:03 2017 ROUTER_CORE (info) In-process subscription
L/$management
Fri May 20 09:38:03 2017 ROUTER_CORE (info) In-process subscription
L/$_management_internal
Fri May 20 09:38:03 2017 DISPLAYNAME (info) Activating
DisplayNameService on $displayname
Fri May 20 09:38:03 2017 ROUTER_CORE (info) In-process subscription
L/$displayname
Fri May 20 09:38:03 2017 CONN_MGR (info) Configured Listener:
0.0.0.0:amqp proto=any role=normal ❹
Fri May 20 09:38:03 2017 POLICY (info) Policy configured
maximumConnections: 0, policyFolder: '', access rules enabled:
'false'
Fri May 20 09:38:03 2017 POLICY (info) Policy fallback
defaultApplication is disabled
Fri May 20 09:38:03 2017 SERVER (info) Operational, 4 Threads
Running ❺

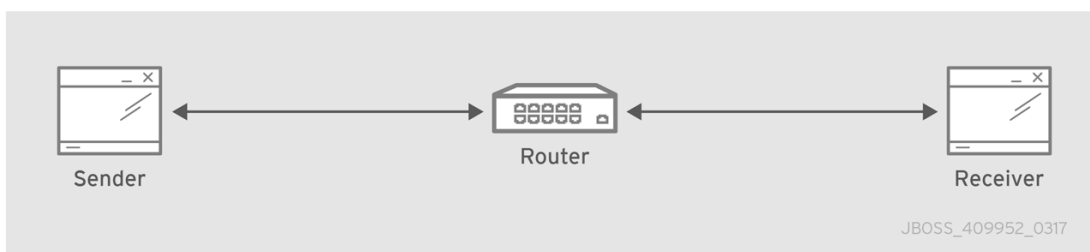
```

- ❶ The name of this router instance.
- ❷ By default, the router starts in *standalone* mode, which means that it cannot connect to other routers or be used in a router network.
- ❸ The management agent. It provides the **\$management** address, through which management tools such as **qdmange** and **qdstat** can perform create, read, update, and delete (CRUD) operations on the router. As an AMQP endpoint, the management agent supports all operations defined by the [AMQP management specification \(Draft 9\)](#).
- ❹ A listener is started on all available network interfaces and listens for connections on the standard AMQP port (5672, which is not encrypted).
- ❺ Threads for handling message traffic and all other internal operations.

## 4.2. ROUTING MESSAGES IN A PEER-TO-PEER CONFIGURATION

This example demonstrates how the router can connect clients by receiving and sending messages between them. It uses the router's default configuration file and does not require a broker.

**Figure 4.1. Peer-to-peer Communication**



As the diagram indicates, the configuration consists of an AMQ Interconnect component with two clients connected to it: a sender and a receiver. The receiver wants to receive messages on a specific address, and the sender sends messages to that address.

A broker is not used in this example, so there is no *"store and forward"* mechanism in the middle. Instead, the messages flow from sender to receiver only if the receiver is online, and the sender can confirm that the messages have arrived at their destination.

This example uses a AMQ Python client to start a receiver client, and then send five messages from the sender client.

## Prerequisites

AMQ Python must be installed before you can complete the peer-to-peer routing example. For more information, see {ClientAmqpPythonUrl}.

## Procedure

1. [Start the receiver client.](#)
2. [Send messages.](#)

### 4.2.1. Starting the Receiver Client

In this example, the receiver client is started first. This means that the messages will be sent as soon as the sender client is started.



#### NOTE

In practice, the order in which you start senders and receivers does not matter. In both cases, messages will be sent as soon as the receiver comes online.

## Procedure

- To start the receiver by using the Python receiver client, navigate to the Python examples directory and run the **simple\_recv.py** example:

```
$ cd INSTALL_DIR/examples/python/  
$ python simple_recv.py -a 127.0.0.1:5672/examples -m 5
```

This command starts the receiver and listens on the default address (**127.0.0.1:5672/examples**). The receiver is also set to receive a maximum of five messages.

### 4.2.2. Sending Messages

After starting the receiver client, you can send messages from the sender. These messages will travel through the router to the receiver.

## Procedure

- In a new terminal window, navigate to the Python examples directory and run the **simple\_send.py** example:

```
$ cd INSTALL_DIR/examples/python/  
$ python simple_send.py -a 127.0.0.1:5672/examples -m 5
```

This command sends five auto-generated messages to the default address (**127.0.0.1:5672/examples**) and then confirms that they were delivered and acknowledged by the receiver:

```
all messages confirmed
```

The receiver client receives the messages and displays their content:

```
{u'sequence': 1L}  
{u'sequence': 2L}  
{u'sequence': 3L}  
{u'sequence': 4L}  
{u'sequence': 5L}
```

## CHAPTER 5. CONFIGURATION

Before starting AMQ Interconnect, you should understand where the router's configuration file is stored, how the file is structured, and the methods you can use to modify it.

### 5.1. ACCESSING THE ROUTER CONFIGURATION FILE

The router's configuration is defined in the router configuration file. You can access this file to view and modify that configuration.

#### Procedure

- Open the following file: `/etc/qpid-dispatch/qdrouterd.conf`.  
When AMQ Interconnect is installed, `qdrouterd.conf` is installed in this directory by default. When the router is started, it runs with the settings defined in this file.

For more information about the router configuration file (including available entities and attributes), see the [qdrouterd man page](#).

### 5.2. HOW THE ROUTER CONFIGURATION FILE IS STRUCTURED

Before you can make changes to a router configuration file, you should understand how the file is structured.

The configuration file contains sections. A section is a configurable entity, and it contains a set of attribute name-value pairs that define the settings for that entity. The syntax is as follows:

```
sectionName {  
    attributeName: attributeValue  
    attributeName: attributeValue  
    ...  
}
```

### 5.3. METHODS FOR USING PATTERN MATCHING AND WILDCARDS

The router configuration file supports pattern matching and wildcards to enable you to match multiple values for certain attributes. However, the syntax varies based on the type of entity that you are configuring.

#### 5.3.1. Pattern Matching for Addresses

In some router configuration scenarios, you might need to use pattern matching to match a range of addresses rather than a single, literal address. Address patterns match any address that corresponds to the pattern.

An address pattern is a sequence of tokens (typically words) that are delimited by either `.` or `/` characters. They also can contain special wildcard characters that represent words:

- `*` represents exactly one word
- `#` represents zero or more words

#### Example 5.1. Address Pattern

This address contains two tokens, separated by the / delimiter:

**my/address**

### Example 5.2. Address Pattern with Wildcard

This address contains three tokens. The \* is a wildcard, representing any single word that might be between **my** and **address**:

**my/\*/address**

The following table shows some address patterns and examples of the addresses that would match them:

This pattern...	Matches...	But not...
<b>news/*</b>	<b>news/europe</b> <b>news/usa</b>	<b>news</b> <b>news/usa/sports</b>
<b>news/#</b>	<b>news</b> <b>news/europe</b> <b>news/usa/sports</b>	<b>europe</b> <b>usa</b>
<b>news/europe/#</b>	<b>news/europe</b> <b>news/europe/sports</b> <b>news/europe/politics/fr</b>	<b>news/usa</b> <b>europe</b>
<b>news/*/sports</b>	<b>news/europe/sports</b> <b>news/usa/sports</b>	<b>news</b> <b>news/europe/fr/sports</b>

### 5.3.2. Pattern Matching for Vhost Policy Hostnames

In a vhost policy, vhost hostnames can be either literal hostnames or patterns that cover a range of hostnames.

A hostname pattern is a sequence of words with one or more of the following wildcard characters:

- \* represents exactly one word
- # represents zero or more words

The following table shows some examples of hostname patterns:

This pattern...	Matches...	But not...
<code>*.example.com</code>	<code>www.example.com</code>	<code>example.comsrv2.www.example.com</code>
<code>#.example.com</code>	<code>example.comwww.example.com</code> <code>a.b.c.d.example.com</code>	<code>myhost.com</code>
<code>www.*.test.example.com</code>	<code>www.a.test.example.com</code>	<code>www.test.example.comwww.a.b.c.test.example.com</code>
<code>www.#.test.example.com</code>	<code>www.test.example.comwww.a.test.example.comwww.a.b.c.test.example.com</code>	<code>test.example.com</code>

Vhost hostname pattern matching applies the following precedence rules:

Policy pattern	Precedence
Exact match	High
<code>*</code>	Medium
<code>#</code>	Low



## NOTE

AMQ Interconnect does not permit you to create vhost hostname patterns that conflict with existing patterns. This includes patterns that can be reduced to be the same as an existing pattern. For example, you would not be able to create the `###.com` pattern if `#.com` already exists.

## 5.4. CHANGING A ROUTER'S CONFIGURATION

You can use different methods for changing a router's configuration based on whether the router is currently running, and whether you want the change to take effect immediately.

### Choices

- [Make a permanent change to the router's configuration.](#)
- [Change the configuration for a running router.](#)

### 5.4.1. Making a Permanent Change to the Router's Configuration

You can make a permanent change to the router's configuration by editing the router's configuration file directly. You must restart the router for the changes to take effect, but the changes will be saved even if the router is stopped.



## Procedure

1. Do one of the following:

- Edit the default configuration file (`/etc/qpid-dispatch/qdrouterd.conf`).
- Create a new configuration file.

2. Start (or restart) the router.

If you created a new configuration file, you must specify the path using the `--conf` parameter. For example, the following command starts the router with a non-default configuration file:

```
$ sudo qdrouterd -d --conf /etc/qpid-dispatch/new-configuration-
file.conf
```

### 5.4.2. Changing the Configuration for a Running Router

If the router is running, you can change its configuration on the fly. The changes you make take effect immediately, but are lost if the router is stopped.

## Procedure

- Use **qdmmanage** to change the configuration.

For more information about using **qdmmanage**, see [Managing AMQ Interconnect Using qdmmanage](#).

## 5.5. DEFAULT CONFIGURATION SETTINGS

The router's configuration file controls the way in which the router functions. The default configuration file contains the minimum number of settings required for the router to run. As you become more familiar with the router, you can add to or change these settings, or create your own configuration files.

When you installed AMQ Interconnect, the default configuration file was added at the following path: `/etc/qpid-dispatch/qdrouterd.conf`. It includes some basic configuration settings that define the router's operating mode, how it listens for incoming connections, and routing patterns for the message routing mechanism.

### Default Configuration File

```
router {
  mode: standalone 1
  id: Router.A 2
}

listener { 3
  host: 0.0.0.0 4
  port: amqp 5
  authenticatePeer: no 6
}

address { 7
  prefix: closest
  distribution: closest
}
```

```

address {
    prefix: multicast
    distribution: multicast
}

address {
    prefix: unicast
    distribution: closest
}

address {
    prefix: exclusive
    distribution: closest
}

address {
    prefix: broadcast
    distribution: multicast
}

```

- 1 By default, the router operates in *standalone* mode. This means that it can only communicate with endpoints that are directly connected to it. It cannot connect to other routers, or participate in a router network.
- 2 The unique identifier of the router. This ID is used as the **container-id** (container name) at the AMQP protocol level. It is required, and the router will not start if this attribute is not defined.
- 3 The **listener** entity handles incoming connections from client endpoints.
- 4 The IP address on which the router will listen for incoming connections. By default, the router is configured to listen on all network interfaces.
- 5 The port on which the router will listen for incoming connections. By default, the default AMQP port (5672) is specified with a symbolic service name.
- 6 Specifies whether the router should authenticate peers before they can connect to the router. By default, peer authentication is not required.
- 7 By default, the router is configured to use the message routing mechanism. Each **address** entity defines how messages that are received with a particular address **prefix** should be distributed. For example, all messages with addresses that start with **closest** will be distributed using the **closest** distribution pattern.



#### NOTE

If a client requests a message with an address that is not defined in the router's configuration file, the **balanced** distribution pattern will be used automatically.

## 5.6. SETTING ESSENTIAL CONFIGURATION PROPERTIES

The router's default configuration settings enable the router to run with minimal configuration. However, you may need to change some of these settings for the router to run properly in your environment.

## Procedure

1. Open the router's configuration file.  
If you are changing the router's default configuration file, the file is located at **/etc/qpid-dispatch/qdrouterd.conf**.
2. To define essential router information, change the following attributes as needed in the **router** section:

```
router {
    mode: STANDALONE/INTERIOR
    id: ROUTER_ID
}
```

### mode

Specify one of the following modes:

- **standalone** - Use this mode if the router does not communicate with other routers and is not part of a router network. When operating in this mode, the router only routes messages between directly connected endpoints.
- **interior** - Use this mode if the router is part of a router network and needs to collaborate with other routers.

### id

The unique identifier for the router. This ID will also be the container name at the AMQP protocol level.

For information about additional attributes, see [router](#) in the **qdrouterd.conf** man page.

3. If necessary for your environment, secure the router.
  - [Set up SSL/TLS for encryption, authentication, or both](#)
  - [Set up SASL for authentication and payload encryption](#)
4. Connect the router to other routers, clients, and brokers.
  - [Add incoming connections](#)
  - [Add outgoing connections](#)
5. Set up routing for your environment:
  - [Configure the router to route messages between clients directly](#)
  - [Configure the router to route messages through a broker queue](#)
  - [Create a link route to define a private messaging path between endpoints](#)
6. [Set up logging](#).

## CHAPTER 6. NETWORK CONNECTIONS

Connections define how the router communicates with clients, other routers, and brokers. You can configure *incoming connections* to define how the router listens for data from clients and other routers, and you can configure *outgoing connections* to define how the router sends data to other routers and brokers.

### 6.1. LISTENING FOR INCOMING CONNECTIONS

Listening for incoming connections involves setting the host and port on which the router should listen for traffic.

#### Procedure

1. In the router's configuration file, add a **listener**:

```
listener {
    host: HOST_NAME/ADDRESS
    port: PORT_NUMBER/NAME
    ...
}
```

#### host

Either an IP address (IPv4 or IPv6) or hostname on which the router should listen for incoming connections.

#### port

The port number or symbolic service name on which the router should listen for incoming connections.

For information about additional attributes, see [listener](#) in the **qdrouterd.conf** man page.

2. If necessary, [secure the connection](#).  
If you have set up SSL/TLS or SASL in your environment, you can configure the router to only accept encrypted or authenticated communication on this connection.
3. If you want the router to listen for incoming connections on additional hosts or ports, configure an additional **listener** entity for each host and port.

### 6.2. ADDING OUTGOING CONNECTIONS

Configuring outgoing connections involves setting the host and port on which the router connects to other routers and brokers.

When a router connects to a broker, the broker might provide backup connection data that the router can use if the primary connection fails. If the primary connection fails, the router attempts to reconnect by using a combination of the primary and — if provided — backup connections in round-robin fashion until the connection is successful. For more information about viewing the backup connection data provided by the broker, see [Section 10.3.2.2, “Managing Connectors”](#).

#### Procedure

1. In the router's configuration file, add a **connector**:

■

```
connector {  
    name: NAME  
    host: HOST_NAME/ADDRESS  
    port: PORT_NUMBER/NAME  
    ...  
}
```

**name**

The name of the **connector**. You should specify a name that describes the entity to which the connector connects. This name is used by configured addresses (for example, a **linkRoute** entity) in order to specify which connection should be used for them.

**host**

Either an IP address (IPv4 or IPv6) or hostname on which the router should connect.

**port**

The port number or symbolic service name on which the router should connect.

For information about additional attributes, see [connector](#) in the **qdrouterd.conf** man page.

2. If necessary, [secure the connection](#).

If you have set up SSL/TLS or SASL in your environment, you can configure the router to only send encrypted or authenticated communication on this connection.

3. For each remaining router or broker to which this router should connect, configure an additional **connector** entity.

## CHAPTER 7. SECURITY

Securing your router network involves configuring authentication and authorization. You can authenticate and encrypt the router's connections using SSL/TLS or SASL. Additionally, you can authorize access to messaging resources by setting user connection restrictions and defining AMQP resource access control.

### 7.1. AUTHENTICATING REMOTE PEERS

You can configure AMQ Interconnect to communicate with clients, routers, and brokers in a secure way by authenticating and encrypting the router's connections. AMQ Interconnect supports the following security protocols:

- *SSL/TLS* for certificate-based encryption and mutual authentication
- *SASL* for authentication and payload encryption

#### 7.1.1. Setting Up SSL/TLS for Encryption and Authentication

Before you can secure incoming and outgoing connections using SSL/TLS encryption and authentication, you must first set up the SSL/TLS profile in the router's configuration file.

##### Prerequisites

You must have the following files in PEM format:

- An X.509 CA certificate (used for signing the router certificate for the SSL/TLS server authentication feature).
- A private key (with or without password protection) for the router.
- An X.509 router certificate signed by the X.509 CA certificate.

##### Procedure

- In the router's configuration file, add an **sslProfile** section:

```
sslProfile {
    name: NAME
    ciphers: CIPHERS
    protocols: PROTOCOL
    caCertFile: PATH.pem
    certFile: PATH.pem
    privateKeyFile: PATH.pem
    password: PASSWORD/PATH_TO_PASSWORD_FILE
    ...
}
```

##### name

A name for the SSL/TLS profile. You can use this name to refer to the profile from the incoming and outgoing connections.

For example:

```
name: router-ssl-profile
```

## ciphers

The SSL cipher suites that can be used by this SSL/TLS profile. If certain ciphers are unsuitable for your environment, you can use this attribute to restrict them from being used. To enable a cipher list, enter one or more cipher strings separated by colons (:).

### Example 7.1. Enabling a Cipher List

```
ciphers:
ALL:!aNULL:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP
```

To see the full list of available ciphers, use the **openssl ciphers** command. For more information about each cipher, see the [ciphers man page](#).

## protocols

The SSL/TLS protocols that this router can use. You can specify a list of one or more of the following values: TLSv1, TLSv1.1, or TLSv1.2.

To specify multiple protocols, separate the protocols with a space.

### Example 7.2. Specifying Multiple Protocols

This example permits the SSL/TLS profile to use TLS v1.1 and TLS v1.2 only:

```
protocols: TLSv1.1 TLSv1.2
```

If you do not specify a value, the router will use the TLS protocol specified by the system-wide configuration.



### NOTE

When setting the TLS protocol versions for the router, you should also consider the TLS protocol version (or versions) used by your client applications. If a subset of TLS protocol versions does not exist between a client and the router, the client will not be able to connect to the router.

## caCertFile

The absolute path to the file that contains the public certificates of trusted certificate authorities (CA).

For example:

```
caCertFile: /qdrouterd/ssl_certs/ca-cert.pem
```

## certFile

The absolute path to the file containing the PEM-formatted public certificate to be used on the local end of any connections using this profile.

For example:

```
certFile: /qdrouterd/ssl_certs/router-cert-pwd.pem
```

**privateKeyFile**

The absolute path to the file containing the PEM-formatted private key for the above certificate.

For example:

```
privateKeyFile: /qdrouterd/ssl_certs/router-key-pwd.pem
```

**passwordFile or password**

If the private key is password-protected, you must provide the password by either specifying the absolute path to a file containing the password that unlocks the certificate key, or entering the password directly in the configuration file.

For example:

```
password: routerKeyPassword
```

For information about additional **sslProfile** attributes, see [sslProfile](#) in the **qdrouterd.conf** man page.

**7.1.2. Setting Up SASL for Authentication and Payload Encryption**

If you plan to use SASL to authenticate connections, you must first add the SASL attributes to the **router** entity in the router's configuration file. These attributes define a set of SASL parameters that can be used by the router's incoming and outgoing connections.

**Prerequisites**

Before you can set up SASL, you must have the following:

- [The SASL database is generated.](#)
- [The SASL configuration file is configured.](#)
- The Cyrus SASL plugin is installed for each SASL mechanism you plan to use. Cyrus SASL uses plugins to support specific SASL mechanisms. Before you can use a particular SASL mechanism, the relevant plugin must be installed. For example, you need the **cyrus-sasl-plain** plugin to use SASL PLAIN authentication.

To see a list of Cyrus SASL plugins in Red Hat Enterprise Linux, use the **yum search cyrus-sasl** command. To install a Cyrus SASL plugin, use the **yum install PLUGIN** command.

**Procedure**

- In the router's configuration file, add the following attributes to the **router** section:

```
router {
    ...
    saslConfigDir: PATH
    saslConfigName: FILE_NAME
}
```

**saslConfigDir**

The absolute path to the SASL configuration file.



For example:

```
saslConfigDir: /qdrouterd/security
```

### **saslConfigName**

The name of the SASL configuration file. This name should *not* include the **.conf** file extension.

For example:

```
saslConfigName: qdrouterd_sasl
```

## 7.1.3. Securing Incoming Connections

You can secure incoming connections by configuring each connection's **listener** entity for encryption, authentication, or both.

### Prerequisites

Before securing incoming connections, the security protocols you plan to use should be set up.

### Choices

- [Add SSL/TLS encryption](#)
- [Add SASL authentication](#)
- [Add SSL/TLS client authentication](#)
- [Add SASL payload encryption](#)

### 7.1.3.1. Adding SSL/TLS Encryption to an Incoming Connection

You can configure an incoming connection to accept encrypted connections only. By adding SSL/TLS encryption, to connect to this router, a remote peer must first start an SSL/TLS handshake with the router and be able to validate the server certificate received by the router during the handshake.

### Procedure

- In the router's configuration file, add the following attributes to the connection's **listener** entity:

```
listener {
    ...
    sslProfile: SSL_PROFILE_NAME
    requireSsl: yes
}
```

### **sslProfile**

The name of the SSL/TLS profile you set up.

### **requireSsl**

Enter **yes** to require all clients connecting to the router on this connection to use encryption.

### 7.1.3.2. Adding SASL Authentication to an Incoming Connection

You can configure an incoming connection to authenticate the client using SASL. You can use SASL authentication with or without SSL/TLS encryption.

#### Procedure

- In the router's configuration file, add the following attributes to the connection's **listener** section:

```
listener {  
    ...  
    authenticatePeer: yes  
    saslMechanisms: MECHANISMS  
}
```

#### **authenticatePeer**

Set this attribute to **yes** to require the router to authenticate the identity of a remote peer before it can use this incoming connection.

#### **saslMechanisms**

The SASL authentication mechanism (or mechanisms) to use for peer authentication. You can choose any of the Cyrus SASL authentication mechanisms *except* for **ANONYMOUS**. To specify multiple authentication mechanisms, separate each mechanism with a space. For a full list of supported Cyrus SASL authentication mechanisms, see [Authentication Mechanisms](#).

### 7.1.3.3. Adding SSL/TLS Client Authentication to an Incoming Connection

You can configure an incoming connection to authenticate the client using SSL/TLS.

The base SSL/TLS configuration provides content encryption and server authentication, which means that remote peers can verify the router's identity, but the router cannot verify a peer's identity.

However, you can require an incoming connection to use SSL/TLS client authentication, which means that remote peers must provide an additional certificate to the router during the SSL/TLS handshake. By using this certificate, the router can verify the client's identity without using a username and password.

You can use SSL/TLS client authentication with or without SASL authentication.

#### Procedure

- In the router's configuration, file, add the following attribute to the connection's **listener** entity:

```
listener {  
    ...  
    authenticatePeer: yes  
}
```

#### **authenticatePeer**

Set this attribute to **yes** to require the router to authenticate the identity of a remote peer before it can use this incoming connection.

### 7.1.3.4. Adding SASL Payload Encryption to an Incoming Connection

If you do not use SSL/TLS, you can still encrypt the incoming connection by using SASL payload encryption.

#### Procedure

- In the router's configuration file, add the following attributes to the connection's **listener** section:

```
listener {
    ...
    requireEncryption: yes
    saslMechanisms: MECHANISMS
}
```

#### **requireEncryption**

Set this attribute to **yes** to require the router to use SASL payload encryption for the connection.

#### **saslMechanisms**

The SASL mechanism to use. You can choose any of the Cyrus SASL authentication mechanisms. To specify multiple authentication mechanisms, separate each mechanism with a space.

For a full list of supported Cyrus SASL authentication mechanisms, see [Authentication Mechanisms](#).

### 7.1.4. Securing Outgoing Connections

You can secure outgoing connections by configuring each connection's **connector** entity for encryption, authentication, or both.

#### Prerequisites

Before securing outgoing connections, the security protocols you plan to use should be set up.

#### Choices

- [Add SSL/TLS authentication](#)
- [Add SASL authentication](#)

#### 7.1.4.1. Adding SSL/TLS Client Authentication to an Outgoing Connection

If an outgoing connection connects to an external client configured with mutual authentication, you should ensure that the outgoing connection is configured to provide the external client with a valid security certificate during the SSL/TLS handshake.

You can use SSL/TLS client authentication with or without SASL authentication.

#### Procedure

- In the router's configuration file, add the **sslProfile** attribute to the connection's **connector** entity:

```
connector {  
    ...  
    sslProfile: SSL_PROFILE_NAME  
}
```

**sslProfile**

The name of the SSL/TLS profile you set up.

### 7.1.4.2. Adding SASL Authentication to an Outgoing Connection

You can configure an outgoing connection to provide authentication credentials to the external container. You can use SASL authentication with or without SSL/TLS encryption.

**Procedure**

- In the router's configuration file, add the **saslMechanisms** attribute to the connection's **connector** entity:

```
connector {  
    ...  
    saslMechanisms: MECHANISMS  
    saslUsername: USERNAME  
    saslPassword: PASSWORD  
}
```

**saslMechanisms**

One or more SASL mechanisms to use to authenticate the router to the external container. You can choose any of the Cyrus SASL authentication mechanisms. To specify multiple authentication mechanisms, separate each mechanism with a space. For a full list of supported Cyrus SASL authentication mechanisms, see [Authentication Mechanisms](#).

**saslUsername**

If any of the SASL mechanisms uses username/password authentication, then provide the username to connect to the external container.

**saslPassword**

If any of the SASL mechanisms uses username/password authentication, then provide the password to connect to the external container.

## 7.2. INTEGRATING WITH KERBEROS

By using the **GSSAPI** SASL mechanism, you can configure AMQ Interconnect to authenticate incoming connections using Kerberos.

**Prerequisites**

- A Kerberos infrastructure must be deployed in your environment.
- In the Kerberos environment, a service principal of **amqp/HOSTNAME@REALM** must be configured.  
This is the service principal that AMQ Interconnect uses.

- The **cyrus-sasl-gssapi** package must be installed on each client and router host machine.
- [SASL must be set up for AMQ Interconnect](#).

## Procedure

1. On the router's host machine, open the **/etc/sasl2/qdrouterd.conf** configuration file.

### Example 7.3. An **/etc/sasl2/qdrouterd.conf** Configuration File

```
pwcheck_method: auxprop
auxprop_plugin: sasl_db
sasldb_path: qdrouterd.sasldb
keytab: /etc/krb5.keytab
mech_list: ANONYMOUS DIGEST-MD5 EXTERNAL PLAIN GSSAPI
```

2. Verify the following:
  - The **mech\_list** attribute contains the **GSSAPI** mechanism.
  - The **keytab** attribute points to the location of the keytab file.
3. Open the router's configuration file.
4. For each incoming connection that should use Kerberos for authentication, set the router's **listener** to use the **GSSAPI** mechanism.

### Example 7.4. A **listener** in the Router Configuration File

```
listener {
    ...
    authenticatePeer: yes
    saslMechanisms: GSSAPI
}
```

For more information about these attributes, see [Section 7.1.3.2, “Adding SASL Authentication to an Incoming Connection”](#).

## 7.3. AUTHORIZING ACCESS TO MESSAGING RESOURCES

You can configure *policies* to secure messaging resources in your messaging environment. Policies ensure that only authorized users can access messaging endpoints through the router network, and that the resources on those endpoints are used in an authorized way.

AMQ Interconnect provides the following types of policies:

### Global policies

Settings for the router. A global policy defines the maximum number of incoming user connections for the router (across all messaging endpoints), and defines how the router should use vhost policies.

### Vhost policies

Connection and AMQP resource limits for a messaging endpoint (called an AMQP virtual host, or vhost). A vhost policy defines what a client can access on a messaging endpoint over a particular connection.

The resource limits defined in global and vhost policies are applied to user connections only. The limits do not affect inter-router connections or router connections that are outbound to waypoints.

### 7.3.1. How AMQ Interconnect Enforces Connection and Resource Limits

AMQ Interconnect uses policies to determine whether to permit a connection, and if it is permitted, to apply the appropriate resource limits.

When a client creates a connection to the router, the router first determines whether to allow or deny the connection. This decision is based on the following criteria:

- Whether the connection will exceed the router's global connection limit (defined in the global policy)
- Whether the connection will exceed the vhost's connection limits (defined in the vhost policy that matches the host to which the connection is directed)

If the connection is allowed, the router assigns the user (the authenticated user name from the connection) to a user group, and enforces the user group's resource limits for the lifetime of the connection.

### 7.3.2. Setting Global Connection Limits

You can set the incoming connection limit for the router. This limit defines the total number of concurrent client connections that can be open for this router.

#### Procedure

- In the router configuration file, add a **policy** section and set the **maxConnections**.

```
policy {  
    maxConnections: 10000  
}
```

#### **maxConnections**

This limit is always enforced, even if no other policy settings have been defined. The limit is applied to all incoming connections regardless of remote host, authenticated user, or targeted vhost. The default (and the maximum) value is **65535**.

### 7.3.3. Setting Connection and Resource Limits for Messaging Endpoints

You can define the connection limit and AMQP resource limits for a messaging endpoint by configuring a *vhost policy*. Vhost policies define what resources clients are permitted to access on a messaging endpoint over a particular connection.



## NOTE

A vhost is typically the name of the host to which the client connection is directed. For example, if a client application opens a connection to the **amqp://mybroker.example.com:5672/queue01** URL, the vhost would be **mybroker.example.com**.

You can create vhost policies using either of the following methods:

- [Configure vhost policies directly in the router configuration file](#)
- [Configure vhost policies as JSON files](#)

### 7.3.3.1. Enabling Vhost Policies

You must enable the router to use vhost policies before you can create the policies.

#### Procedure

- In the router configuration file, add a **policy** section if one does not exist, and enable vhost policies for the router.

```
policy {
    ...
    enableVhostPolicy: true
    enableVhostNamePatterns: true | false
    defaultVhost: $default
}
```

#### **enableVhostPolicy**

Enables the router to enforce the connection denials and resource limits defined in the configured vhost policies. The default is **false**, which means that the router will not enforce any vhost policies.

#### **enableVhostNamePatterns**

Enables pattern matching for vhost hostnames. If set to **true**, you can use wildcards to specify a range of hostnames for a vhost. If set to **false**, vhost hostnames are treated as literal strings. This means that you must specify the exact hostname for each vhost. The default is **false**.

#### **defaultVhost**

The name of the default vhost policy, which is applied to any connection for which a vhost policy has not been configured. The default is **\$default**. If **defaultVhost** is not defined, then default vhost processing is disabled.

### 7.3.3.2. Configuring Vhost Policies in the Router Configuration File

You can configure vhost policies in the router configuration file by configuring **vhost** entities. However, if multiple routers in your router network should be configured with the same vhost configuration, you will need to add the vhost configuration to each router's configuration file.

#### Prerequisites

Vhost policies must be enabled for the router. For more information, see [Section 7.3.3.1, "Enabling Vhost Policies"](#).

## Procedure

1. Add a **vhost** section and define the connection limits for the messaging endpoint.

The connection limits apply to all users that are connected to the vhost. These limits control the number of users that can be connected simultaneously to the vhost.

```
vhost {
  hostname: example.com
  maxConnections: 10000
  maxConnectionsPerUser: 100
  maxConnectionsPerHost: 100
  allowUnknownUser: true
  ...
}
```

### hostname

The literal hostname of the vhost (the messaging endpoint) or a pattern that matches the vhost hostname. This vhost policy will be applied to any client connection that is directed to the hostname that you specify. This name must be unique; you can only have one vhost policy per hostname.

If **enableVhostNamePatterns** is set to **true**, you can use wildcards to specify a pattern that matches a range of hostnames. For more information, see [Section 5.3.2, “Pattern Matching for Vhost Policy Hostnames”](#).

### maxConnections

The global maximum number of concurrent client connections allowed for this vhost. The default is 65535.

### maxConnectionsPerUser

The maximum number of concurrent client connections allowed for any user. The default is 65535.

### maxConnectionsPerHost

The maximum number of concurrent client connections allowed for any remote host (the host from which the client is connecting). The default is 65535.

### allowUnknownUser

Whether unknown users (users who are not members of a defined user group) are allowed to connect to the vhost. Unknown users are assigned to the `$default` user group and receive `$default` settings. The default is `false`, which means that unknown users are not allowed.

2. In the **vhost** section, beneath the connection settings that you added, add a **groups** entity to define the resource limits.

You define resource limits by user group. A user group specifies the messaging resources the members of the group are allowed to access.

### Example 7.5. User Groups in a Vhost Policy

This example shows three user groups: `admin`, `developers`, and `$default`:

```
vhost {
  ...
  groups: {
    admin: {
      users: admin1, admin2
```



```

        remoteHosts: 127.0.0.1, ::1
        sources: *
        targets: *
    }
    developers: {
        users: dev1, dev2, dev3
        remoteHosts: *
        sources: myqueue1, myqueue2
        targets: myqueue1, myqueue2
    }
    $default: {
        remoteHosts: *
        allowDynamicSource: true,
        sources: myqueue1, myqueue2
        targets: myqueue1, myqueue2
    }
}
}

```

### users

A list of authenticated users for this user group. Use commas to separate multiple users. A user may belong to only one vhost user group.

### remoteHosts

A list of remote hosts from which the users may connect. A host can be a hostname, IP address, or IP address range. Use commas to separate multiple hosts. To allow access from all remote hosts, specify a wildcard \*. To deny access from all remote hosts, leave this attribute blank.

### sources | sourcePattern

A list of AMQP source addresses from which users in this group may receive messages. Use **sources** to specify one or more literal addresses. To specify multiple addresses, use a comma-separated list. To prevent users in this group from receiving messages from any addresses, leave this attribute blank. To allow access to an address specific to a particular user, specify the **\${user}** token. For more information, see [Section 7.3.3.4, “Methods for Specifying Vhost Policy Source and Target Addresses”](#).

Alternatively, you can use **sourcePattern** to match one or more addresses that correspond to a pattern. A pattern is a sequence of words delimited by either a . or / character. You can use wildcard characters to represent a word. The \* character matches exactly one word, and the # character matches any sequence of zero or more words.

To specify multiple address ranges, use a comma-separated list of address patterns. For more information, see [Router Address Pattern Matching](#). To allow access to address ranges that are specific to a particular user, specify the **\${user}** token. For more information, see [Section 7.3.3.4, “Methods for Specifying Vhost Policy Source and Target Addresses”](#).

### targets | targetPattern

A list of AMQP target addresses from which users in this group may send messages. You can specify multiple AMQP addresses and use user name substitution and address patterns the same way as with source addresses.

A list of AMQP target addresses from which users in this group may send messages. You can specify multiple AMQP addresses and use user name substitution and wildcards the same way as with source addresses.

3. If necessary, add any advanced user group settings to the vhost user groups.  
The advanced user group settings enable you to define resource limits based on the AMQP connection open, session begin, and link attach phases of the connection. For more information, see [vhost](#) in the `qdrouterd.conf` man page.

### 7.3.3.3. Configuring Vhost Policies as JSON Files

As an alternative to using the router configuration file, you can configure vhost policies in JSON files. If you have multiple routers that need to share the same vhost configuration, you can put the vhost configuration JSON files in a location accessible to each router, and then configure the routers to apply the vhost policies defined in these JSON files.

#### Prerequisites

- Vhost policies must be enabled for the router. For more information, see [Section 7.3.3.1, “Enabling Vhost Policies”](#).

#### Procedure

1. In the router configuration file, specify the directory where you want to store the vhost policy definition JSON files.

```
policy {
    ...
    policyDir: DIRECTORY_PATH
}
```

#### **policyDir**

The absolute path to the directory that holds vhost policy definition files in JSON format. The router processes all of the vhost policies in each JSON file that is in this directory.

2. In the vhost policy definition directory, create a JSON file for each vhost policy.

#### **Example 7.6. Vhost Policy Definition JSON File**

```
[
  [
    "vhost", {
      "hostname": "example.com",
      "maxConnections": 10000,
      "maxConnectionsPerUser": 100,
      "maxConnectionsPerHost": 100,
      "allowUnknownUser": true,
      "groups": {
        "admin": {
          "users": ["admin1", "admin2"],
          "remoteHosts": ["127.0.0.1", ":::1"],
          "sources": "*",
          "targets": "*"
        },
        "developers": {
```

```

    "users": ["dev1", "dev2", "dev3"],
    "remoteHosts": "*",
    "sources": ["myqueue1", "myqueue2"],
    "targets": ["myqueue1", "myqueue2"]
  },
  "$default": {
    "remoteHosts": "*",
    "allowDynamicSource": true,
    "sources": ["myqueue1", "myqueue2"],
    "targets": ["myqueue1", "myqueue2"]
  }
}
}]
]
```



For more information about these attributes, see [Section 7.3.3.2, “Configuring Vhost Policies in the Router Configuration File”](#).

7.3.3.4. Methods for Specifying Vhost Policy Source and Target Addresses

If you want to allow or deny access to multiple addresses on a vhost, there are several methods you can use to match multiple addresses without having to specify each address individually.

The following table describes the methods you can use to specify multiple source and target addresses for a vhost:

To...	Do this...
Allow all users in the user group to access all source or target addresses on the vhost	Use a * wildcard character.  Example 7.7. Receive from Any Address  sources: *
Prevent all users in the user group from accessing all source or target addresses on the vhost	Do not specify a value.  Example 7.8. Prohibit Message Transfers to All Addresses  targets:

To...	Do this...
Allow access to some resources specific to each user	<p>Use the <b><code>\${user}</code></b> username substitution token. You can use this token with <b><code>source</code></b>, <b><code>target</code></b>, <b><code>sourcePattern</code></b>, and <b><code>targetPattern</code></b>.</p> <div>  <div> <p><b>NOTE</b></p> <p>You can only specify the <b><code>\${user}</code></b> token once in an AMQP address name or pattern. If there are multiple tokens in an address, only the leftmost token will be substituted.</p> </div> </div> <p><b>Example 7.9. Receive from a User-Specific Address</b></p> <p>This definition allows the users in the user group to receive messages from any address that meets any of the following rules:</p> <ul style="list-style-type: none"> <li>• Starts with the prefix <b><code>tmp_</code></b> and ends with the user name</li> <li>• Starts with the prefix <b><code>temp</code></b> followed by any additional characters</li> <li>• Starts with the user name, is followed by <b><code>-home-</code></b>, and ends with any additional characters</li> </ul> <pre>sources: tmp_\${user}, temp*, \${user}-home-*</pre> <p><b>Example 7.10. User-Specific Address Patterns</b></p> <p>This definition allows the users in the user group to receive messages from any address that meets any of the following rules:</p> <ul style="list-style-type: none"> <li>• Starts with the prefix <b><code>tmp</code></b> and ends with the user name</li> <li>• Starts with the prefix <b><code>temp</code></b> followed by zero or more additional characters</li> <li>• Starts with the user name, is followed by <b><code>home</code></b>, and ends with one or more additional characters</li> </ul> <pre>sourcePattern: tmp.\${user}, temp/#, \${user}.home/*</pre> <div>  <div> <p><b>NOTE</b></p> <p>In an address pattern (<b><code>sourcePattern</code></b> or <b><code>targetPattern</code></b>), the username substitution token must be either the first or last token in the pattern. The token must also be alone within its delimited field, which means that it cannot be concatenated with literal text prefixes or suffixes.</p> </div> </div>

### 7.3.3.5. Vhost Policy Examples

These examples demonstrate how to use vhost policies to authorize access to messaging resources.

### Example 7.11. Defining Basic Resource Limits for a Messaging Endpoint

In this example, a vhost policy defines resource limits for clients connecting to the **example.com** host.

```
[
  [ "vhost", {
    "hostname": "example.com",
    "maxConnectionsPerUser": 10,
    "allowUnknownUser": true,
    "groups": {
      "admin": {
        "users": ["admin1", "admin2"],
        "remoteHosts": ["127.0.0.1", "::1"],
        "sources": "*",
        "targets": "*"
      },
      "$default": {
        "remoteHosts": "*",
        "sources": ["news*", "sports*", "chat*"],
        "targets": "chat*"
      }
    }
  } ]
]
```

- 1 The rules defined in this vhost policy will be applied to any user connecting to **example.com**.
- 2 Each user can open up to 10 connections to the vhost.
- 3 Any user can connect to this vhost. Users that are not part of the **admin** group are assigned to the **\$default** group.
- 4 If the **admin1** or **admin2** user connects to the vhost, they are assigned to the **admin** user group.
- 5 Users in the **admin** user group must connect from localhost. If the admin user attempts to connect from any other host, the connection will be denied.
- 6 Users in the admin user group can receive from any address offered by the vhost.
- 7 Users in the admin user group can send to any address offered by the vhost.
- 8 Any non-admin user is permitted to connect from any host.
- 9 Non-admin users are permitted to receive messages from any addresses that start with the **news**, **sports**, or **chat** prefixes.
- 10 Non-admin users are permitted to send messages to any addresses that start with the **chat** prefix.

### Example 7.12. Limiting Memory Consumption

By using the advanced vhost policy attributes, you can control how much system buffer memory a user connection can potentially consume.

In this example, a stock trading site provides services for stock traders. However, the site must also accept high-capacity, automated data feeds from stock exchanges. To prevent trading activity from consuming memory needed for the feeds, a larger amount of system buffer memory is allotted to the feeds than to the traders.

This example uses the **maxSessions** and **maxSessionWindow** attributes to set the buffer memory consumption limits for each AMQP session. These settings are passed directly to the AMQP connection and session negotiations, and do not require any processing cycles on the router.

This example does not show the vhost policy settings that are unrelated to buffer allocation.

```
[
  [ "vhost", {
    "hostname": "traders.com",
    "groups": {
      "traders": {
        "users": ["trader1", "trader2"],
        "maxFrameSize": 10000,
        "maxSessionWindow": 5000000,
        "maxSessions": 1
      },
      "feeds": {
        "users": ["nyse-feed", "nasdaq-feed"],
        "maxFrameSize": 60000,
        "maxSessionWindow": 1200000000,
        "maxSessions": 3
      }
    }
  } ]
]
```

- 1 The rules defined in this vhost policy will be applied to any user connecting to **traders.com**.
- 2 The **traders** group includes **trader1**, **trader2**, and any other user defined in the list.
- 3 At most, 5,000,000 bytes of data can be in flight on each session.
- 4 Only one session per connection is allowed.
- 5 The **feeds** group includes two users.
- 6 At most, 1,200,000,000 bytes of data can be in flight on each session.
- 7 Up to three sessions per connection are allowed.

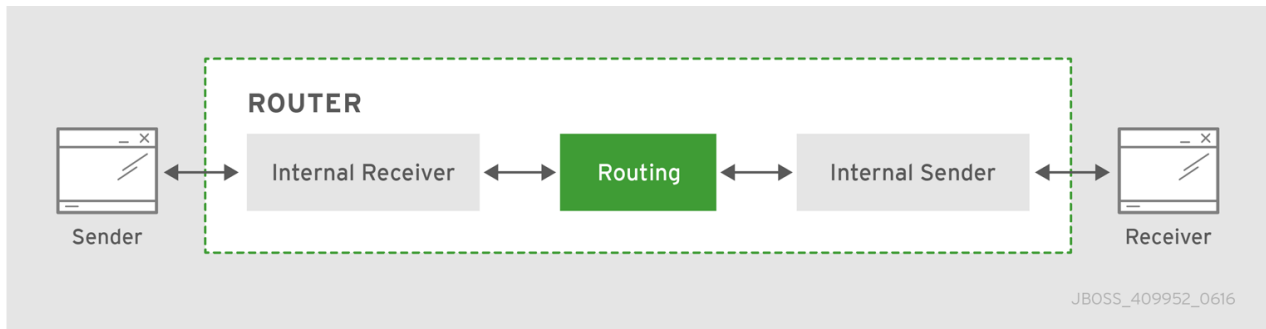
## CHAPTER 8. ROUTING

Routing is the process by which messages are delivered to their destinations. To accomplish this, AMQ Interconnect provides two routing mechanisms: *message routing* and *link routing*.

### Message routing

Routing is performed on messages as producers send them to a router. When a message arrives on a router, the router routes the message and its *settlement* based on the message's *address* and *routing pattern*.

**Figure 8.1. Message Routing**

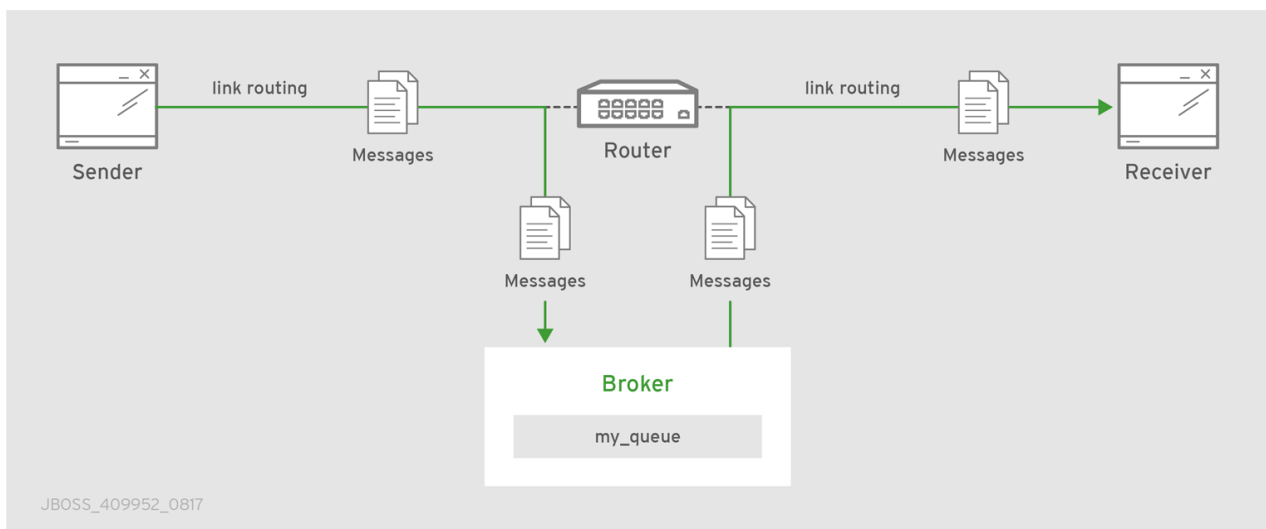


In this diagram, the message producer attaches a link to the router, and then sends a message over the link. When the router receives the message, it identifies the message's destination based on the message's address, and then uses its routing table to determine the best route to deliver the message either to its destination or to the next hop in the route. All dispositions (including settlement) are propagated along the same path that the original message transfer took. Flow control is handled between the sender and the router, and then between the router and the receiver.

### Link routing

Routing is performed on link-attach frames, which are chained together to form a virtual messaging path that directly connects a sender and receiver. Once a link route is established, the transfer of message deliveries, flow frames, and dispositions is performed across the link route.

**Figure 8.2. Link Routing**



In this diagram, a router is connected to clients and to a broker, and it provides a link route to a queue on the broker (*my\_queue*). The sender connects to the router, and the router propagates the link-attaches to the broker to form a direct link between the sender and the broker. The sender can begin

sending messages to the queue, and the router passes the deliveries along the link route directly to the broker queue.

## 8.1. COMPARISON OF MESSAGE ROUTING AND LINK ROUTING

While you can use either message routing or link routing to deliver messages to a destination, they differ in several important ways. Understanding these differences will enable you to choose the proper routing approach for any particular use case.

### 8.1.1. When to Use Message Routing

Message routing is the default routing mechanism. You can use it to route messages on a per-message basis between clients directly (direct-routed messaging), or to and from broker queues (brokered messaging).

Message routing is best suited to the following requirements:

- Default, basic message routing.  
AMQ Interconnect automatically routes messages by default, so manual configuration is only required if you want routing behavior that is different than the default.
- Message-based routing patterns.  
Message routing supports both anycast and multicast routing patterns. You can load-balance individual messages across multiple consumers, and multicast (or fan-out) messages to multiple subscribers.
- Sharding messages across multiple broker instances when message delivery order is not important.  
Sharding messages from one producer might cause that producer's messages to be received in a different order than the order in which they were sent.

Message routing is not suitable for any of the following requirements:

- Dedicated path through the router network.  
For inter-router transfers, all message deliveries are placed on the same inter-router link. This means that the traffic for one address might affect the delivery of the traffic for another address.
- Granular, end-to-end flow control.  
With message routing, end-to-end flow control is based on the settlement of deliveries and therefore might not be optimal in every case.
- Transaction support.
- Server-side selectors.

### 8.1.2. When to Use Link Routing

Link routing requires more detailed configuration than message routing as well as an AMQP container that can accept incoming link-attaches (typically a broker). However, link routing enables you to satisfy more advanced use cases than message routing.

You can use link routing if you need to meet any of the following requirements:

- Dedicated path through the router network.



With link routing, each link route has dedicated inter-router links through the network. Each link has its own dedicated message buffers, which means that the address will not have "head-of-line" blocking issues with other addresses.

- Sharding messages across multiple broker instances with guaranteed delivery order.  
Link routing to a sharded queue preserves the delivery order of the producer's messages by causing all messages on that link to go to the same broker instance.
- End-to-end flow control.  
Flow control is "real" in that credits flow across the link route from the receiver to the sender.
- Transaction support.  
Link routing supports local transactions to a single broker. Distributed transactions are not supported.
- Server-side selectors.  
With a link route, consumers can provide server-side selectors for broker subscriptions.

## 8.2. CONFIGURING MESSAGE ROUTING

With message routing, routing is performed on messages as producers send them to a router. When a message arrives on a router, the router routes the message and its *settlement* based on the message's *address* and *routing pattern*.

With message routing, you can do the following:

- Route messages between clients (direct-routed, or brokerless messaging)  
This involves configuring an address with a routing pattern. All messages sent to the address will be routed based on the routing pattern.
- Route messages through a broker queue (brokered messaging)  
This involves configuring a waypoint address to identify the broker queue and then connecting the router to the broker. All messages sent to the waypoint address will be routed to the broker queue.

### 8.2.1. Addresses

Addresses determine how messages flow through your router network. An address designates an endpoint in your messaging network, such as:

- Endpoint processes that consume data or offer a service
- Topics that match multiple consumers to multiple producers
- Entities within a messaging broker:
  - Queues
  - Durable Topics
  - Exchanges

When a router receives a message, it uses the message's address to determine where to send the message (either its destination or one step closer to its destination).

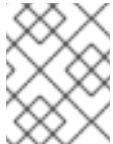
## 8.2.2. Routing Patterns

Each address has one of the following routing patterns, which define the path that a message with the address can take across the messaging network:

### Balanced

An anycast method that allows multiple consumers to use the same address. Each message is delivered to a single consumer only, and AMQ Interconnect attempts to balance the traffic load across the router network.

If multiple consumers are attached to the same address, each router determines which outbound path should receive a message by considering each path's current number of unsettled deliveries. This means that more messages will be delivered along paths where deliveries are settled at higher rates.

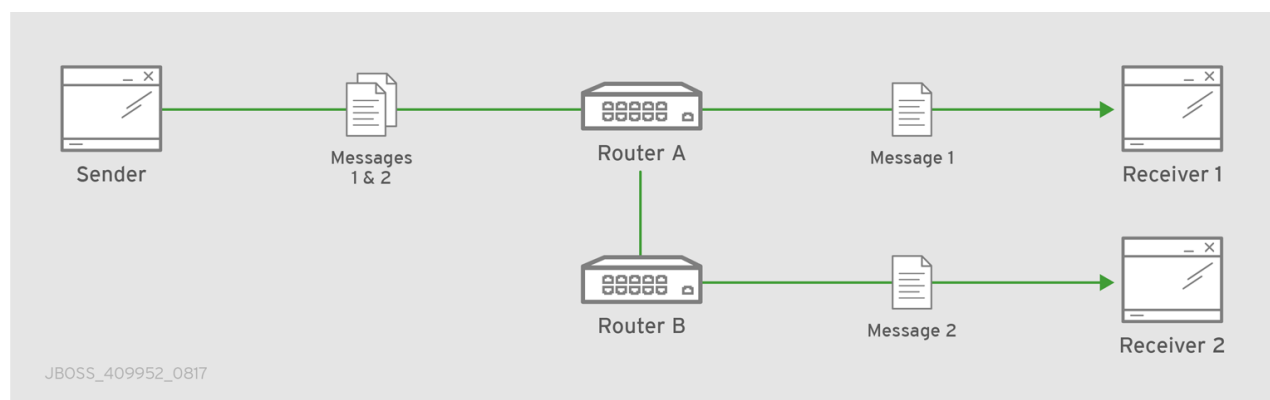


#### NOTE

AMQ Interconnect neither measures nor uses message settlement time to determine which outbound path to use.

In this scenario, the messages are spread across both receivers regardless of path length:

**Figure 8.3. Balanced Message Routing**

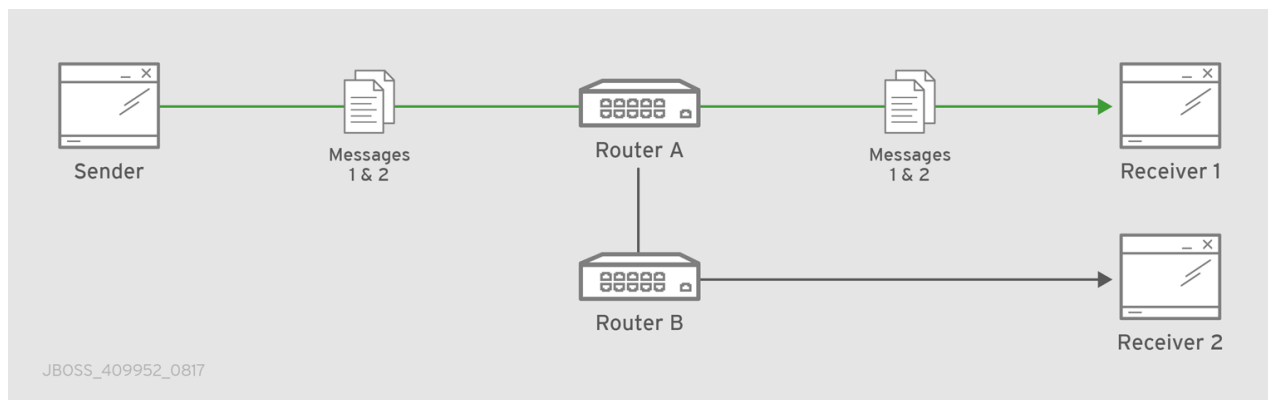


### Closest

An anycast method in which every message is sent along the shortest path to reach the destination, even if there are other consumers for the same address.

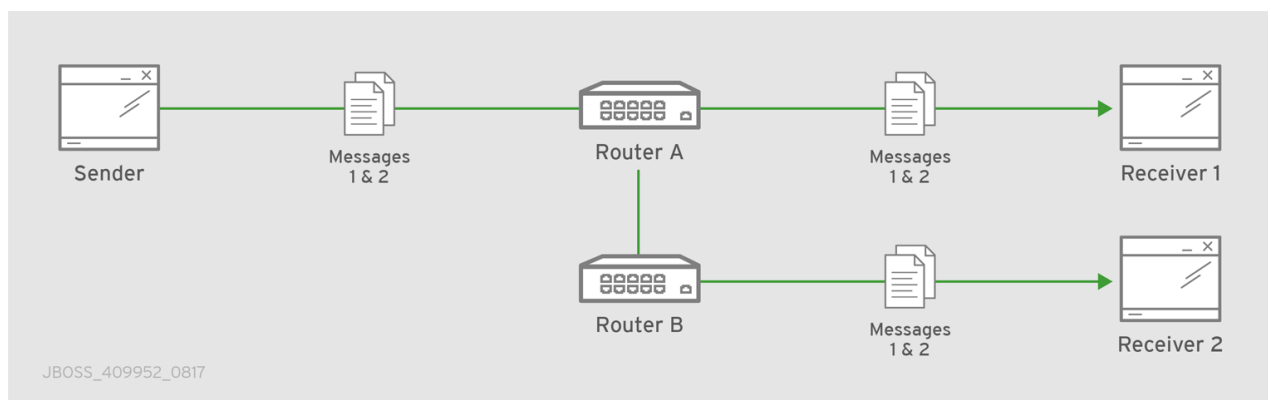
AMQ Interconnect determines the shortest path based on the topology cost to reach each of the consumers. If there are multiple consumers with the same lowest cost, messages will be spread evenly among those consumers.

In this scenario, all messages sent by **Sender** will be delivered to **Receiver 1**:

**Figure 8.4. Closest Message Routing****Multicast**

Messages are sent to all consumers attached to the address. Each consumer will receive one copy of the message.

In this scenario, all messages are sent to all receivers:

**Figure 8.5. Multicast Message Routing****8.2.3. Message Settlement**

Message settlement is negotiated between the producer and the router when the producer establishes a link to the router. Depending on the settlement pattern, messages might be delivered with any of the following degrees of reliability:

- At most once
- At least once
- Exactly once

AMQ Interconnect treats all messages as either *pre-settled* or *unsettled*, and it is responsible for propagating the settlement of each message it routes.

**Pre-settled**

Sometimes called *fire and forget*, the router settles the incoming and outgoing deliveries and propagates the settlement to the message's destination. However, it does not guarantee delivery.

**Unsettled**

The router propagates the settlement between the sender and receiver, and guarantees one of the following outcomes:

- The message is delivered and settled, with the consumer's disposition indicated.
- The delivery is settled with a disposition of **RELEASED**.  
This means that the message did not reach its destination.
- The delivery is settled with a disposition of **MODIFIED**.  
This means that the message might or might not have reached its destination. The delivery is considered to be "in-doubt" and should be re-sent if "at least once" delivery is required.
- The link, session, or connection to AMQ Interconnect was dropped, and all deliveries are "in-doubt".

## 8.2.4. Routing Messages Between Clients

You can route messages between clients without using a broker. In a brokerless scenario (sometimes called *direct-routed messaging*), AMQ Interconnect routes messages between clients directly.

To route messages between clients, you configure an address with a routing distribution pattern. When a router receives a message with this address, the message is routed to its destination or destinations based on the address's routing distribution pattern.

### Procedure

1. In the router's configuration file, add an **address** section:

```
address {  
    prefix: ADDRESS_PREFIX  
    distribution: balanced|closest|multicast  
    ...  
}
```

#### **prefix | pattern**

The address or group of addresses to which the address settings should be applied. You can specify a prefix to match an exact address or beginning segment of an address. Alternatively, you can specify a pattern to match an address using wildcards.

A *prefix* matches either an exact address or the beginning segment within an address that is delimited by either a `.` or `/` character. For example, the prefix **my\_address** would match the address **my\_address** as well as **my\_address.1** and **my\_address/1**. However, it would not match **my\_address1**.

A *pattern* matches an address that corresponds to a pattern. A pattern is a sequence of words delimited by either a `.` or `/` character. You can use wildcard characters to represent a word.

The `*` character matches exactly one word, and the `#` character matches any sequence of zero or more words.

The `*` and `#` characters are reserved as wildcards. Therefore, you should not use them in the message address.

For more information about creating address patterns, see [Section 5.3.1, "Pattern Matching for Addresses"](#).



## NOTE

You can convert a **prefix** value to a **pattern** by appending `/#` to it. For example, the prefix `a/b/c` is equivalent to the pattern `a/b/c/#`.

### distribution

The message distribution pattern. The default is **balanced**, but you can specify any of the following options:

- **balanced** - Messages sent to the address will be routed to one of the receivers, and the routing network will attempt to balance the traffic load based on the rate of settlement.
- **closest** - Messages sent to the address are sent on the shortest path to reach the destination. It means that if there are multiple receivers for the same address, only the closest one will receive the message.
- **multicast** - Messages are sent to all receivers that are attached to the address in a *publish/subscribe* model.

For more information about message distribution patterns, see [Routing Patterns](#).

For information about additional attributes, see [address](#) in the `qdrouterd.conf` man page.

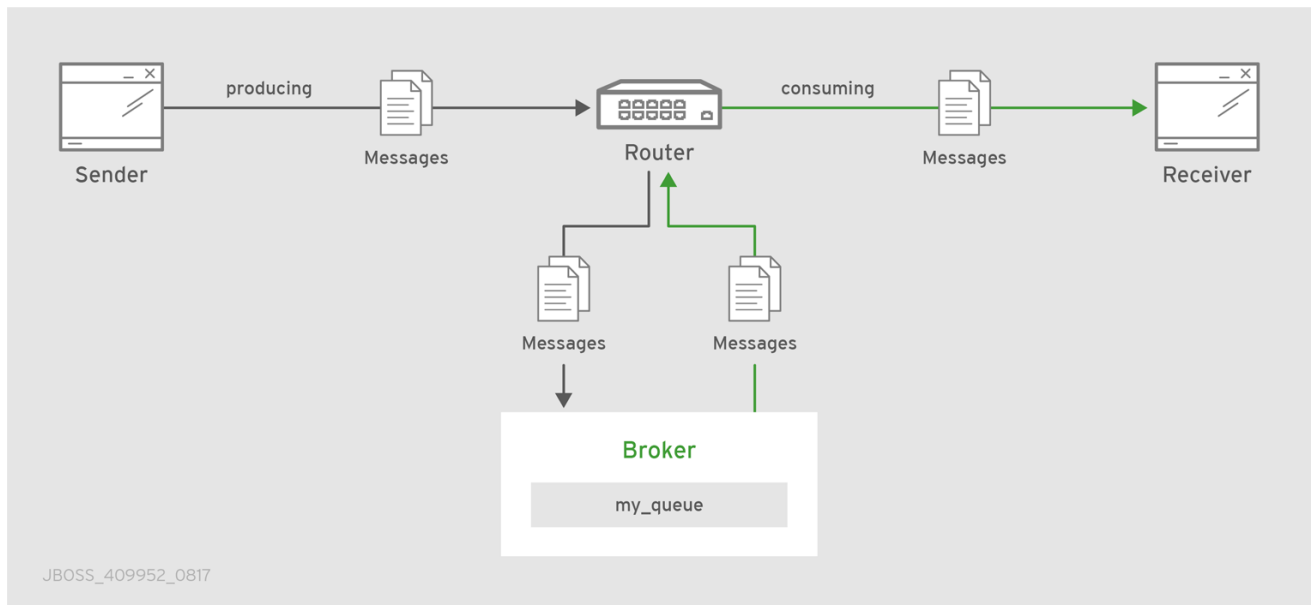
2. Add the same **address** section to any other routers that need to use the address.

The **address** that you added to this router configuration file only controls how this router distributes messages sent to the address. If you have additional routers in your router network that should distribute messages for this address, then you must add the same **address** section to each of their configuration files.

## 8.2.5. Routing Messages Through a Broker Queue

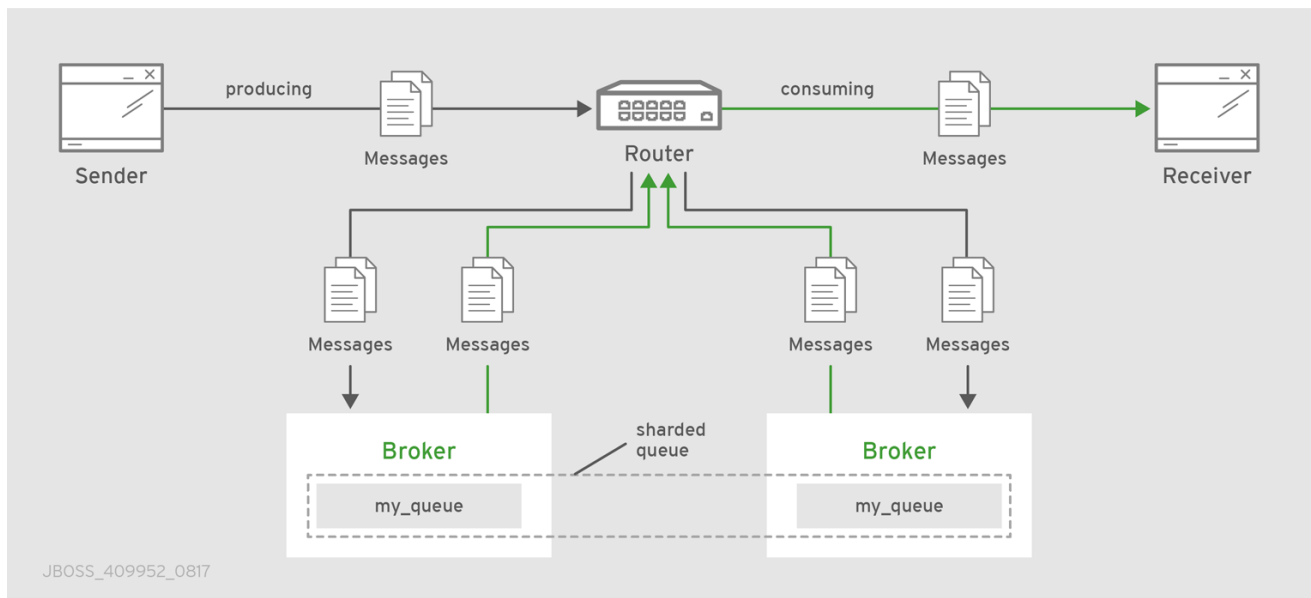
You can route messages to and from a broker queue to provide clients with access to the queue through a router. In this scenario, clients connect to a router to send and receive messages, and the router routes the messages to or from the broker queue.

You can route messages to a queue hosted on a single broker, or route messages to a *sharded queue* distributed across multiple brokers.

**Figure 8.6. Brokered Messaging**

In this diagram, the sender connects to the router and sends messages to `my_queue`. The router attaches an outgoing link to the broker, and then sends the messages to `my_queue`. Later, the receiver connects to the router and requests messages from `my_queue`. The router attaches an incoming link to the broker to receive the messages from `my_queue`, and then delivers them to the receiver.

You can also route messages to a *sharded queue*, which is a single, logical queue comprised of multiple, underlying physical queues. Using queue sharding, it is possible to distribute a single queue over multiple brokers. Clients can connect to any of the brokers that hold a shard to send and receive messages.

**Figure 8.7. Brokered Messaging with Sharded Queue**

In this diagram, a sharded queue (`my_queue`) is distributed across two brokers. The router is connected to the clients and to both brokers. The sender connects to the router and sends messages to `my_queue`. The router attaches an outgoing link to each broker, and then sends messages to each shard (by default, the routing distribution is **balanced**). Later, the receiver connects to the router and requests all of the messages from `my_queue`. The router attaches an incoming link to one of the brokers to receive the messages from `my_queue`, and then delivers them to the receiver.

## Procedure

1. [Add a waypoint address.](#)  
This address identifies the queue to which you want to route messages.
2. [Add autolinks to connect the router to the broker.](#)  
Autolinks connect the router to the broker queue identified by the waypoint address.
3. [If the queue is sharded, add autolinks for each additional broker that hosts a shard](#)

### 8.2.5.1. Configuring Waypoint Addresses

A waypoint address identifies a queue on a broker to which you want to route messages. You need to configure the waypoint address on each router that needs to use the address. For example, if a client is connected to *Router A* to send messages to the broker queue, and another client is connected to *Router B* to receive those messages, then you would need to configure the waypoint address on both *Router A* and *Router B*.

## Prerequisites

An incoming connection (**listener**) to which the clients can connect should be configured. This connection defines how the producers and consumers connect to the router to send and receive messages. For more information, see [Adding Incoming Connections](#).

## Procedure

- Create waypoint addresses on each router that needs to use the address:

```
address {
    prefix: ADDRESS_PREFIX
    waypoint: yes
}
```

### prefix | pattern

The address prefix or pattern that matches the broker queue to which you want to send messages. You can specify a prefix to match an exact address or beginning segment of an address. Alternatively, you can specify a pattern to match an address using wildcards. A *prefix* matches either an exact address or the beginning segment within an address that is delimited by either a `.` or `/` character. For example, the prefix **my\_address** would match the address **my\_address** as well as **my\_address.1** and **my\_address/1**. However, it would not match **my\_address1**.

A *pattern* matches an address that corresponds to a pattern. A pattern is a sequence of words delimited by either a `.` or `/` character. You can use wildcard characters to represent a word. The `*` character matches exactly one word, and the `#` character matches any sequence of zero or more words.

The `*` and `#` characters are reserved as wildcards. Therefore, you should not use them in the message address.

For more information about creating address patterns, see [Section 5.3.1, “Pattern Matching for Addresses”](#).

**NOTE**

You can convert a **prefix** value to a **pattern** by appending `/#` to it. For example, the prefix `a/b/c` is equivalent to the pattern `a/b/c/#`.

**waypoint**

Set this attribute to **yes** so that the router handles messages sent to this address as a waypoint.

**8.2.5.2. Connecting a Router to the Broker**

After you add waypoint addresses to identify the broker queue, you must connect a router to the broker using autolinks.

With autolinks, client traffic is handled on the router, not the broker. Clients attach their links to the router, and then the router uses internal autolinks to connect to the queue on the broker. Therefore, the queue will always have a single producer and a single consumer regardless of how many clients are attached to the router.

**NOTE**

If the connection to the broker fails, AMQ Interconnect automatically attempts to reestablish the connection and reroute message deliveries to any available alternate destinations. However, some deliveries could be returned to the sender with a **RELEASED** or **MODIFIED** disposition. Therefore, you should ensure that your clients can handle these deliveries appropriately (generally by resending them).

1. If this router is different than the router that is connected to the clients, then add the waypoint address.
2. Add an outgoing connection to the broker:

```
connector {
  name: NAME
  host: HOST_NAME/ADDRESS
  port: PORT_NUMBER/NAME
  role: route-container
  ...
}
```

**name**

The name of the **connector**. Specify a name that describes the broker.

**host**

Either an IP address (IPv4 or IPv6) or hostname on which the router should connect to the broker.

**port**

The port number or symbolic service name on which the router should connect to the broker.

**role**

Specify **route-container** to indicate that this connection is for an external container (broker).



For information about additional attributes, see [connector](#) in the **qdrouterd.conf** man page.

3. If you want to send messages to the broker queue, create an outgoing autolink to the broker queue:

```
autoLink {
    addr: ADDRESS
    connection: CONNECTOR_NAME
    direction: out
    ...
}
```

#### **addr**

The address of the broker queue. When the autolink is created, it will be attached to this address.

#### **externalAddr**

An optional alternate address for the broker queue. You use an external address if the broker queue should have a different address than that which the sender uses. In this scenario, senders send messages to the **addr** address, and then the router routes them to the broker queue represented by the **externalAddr** address.

#### **connection | containerID**

How the router should connect to the broker. You can specify either an outgoing connection (**connection**) or the container ID of the broker (**containerID**).

#### **direction**

Set this attribute to **out** to specify that this autolink can send messages from the router to the broker.

For information about additional attributes, see [autoLink](#) in the **qdrouterd.conf** man page.

4. If you want to receive messages from the broker queue, create an incoming autolink from the broker queue:

```
autoLink {
    addr: ADDRESS
    connection: CONNECTOR_NAME
    direction: in
    ...
}
```

#### **addr**

The address of the broker queue. When the autolink is created, it will be attached to this address.

#### **externalAddr**

An optional alternate address for the broker queue. You use an external address if the broker queue should have a different address than that which the receiver uses. In this scenario, receivers receive messages from the **addr** address, and the router retrieves them from the broker queue represented by the **externalAddr** address.

#### **connection | containerID**

How the router should connect to the broker. You can specify either an outgoing connection (**connection**) or the container ID of the broker (**containerID**).

**direction**

Set this attribute to **in** to specify that this autolink can receive messages from the broker to the router.

For information about additional attributes, see [autoLink](#) in the **qdrouterd.conf** man page.

## 8.2.6. Example: Routing Messages Through Broker Queues

This example shows how waypoints and autolinks can route messages through a pair of queues on a broker.

### 8.2.6.1. Router Configuration

```
connector { ❶
    name: broker
    role: route-container
    host: 198.51.100.1
    port: 61617
    saslMechanisms: ANONYMOUS
}

address { ❷
    prefix: queue
    waypoint: yes
}

autoLink { ❸
    addr: queue.first
    direction: in
    connection: broker
}

autoLink { ❹
    addr: queue.first
    direction: out
    connection: broker
}

autoLink { ❺
    addr: queue.second
    direction: in
    connection: broker
}

autoLink { ❻
    addr: queue.second
    direction: out
    connection: broker
}
```

❶ The outgoing connection from the router to the broker. The **route-container** role enables the router to connect to an external AMQP container (in this case, a broker).

❷

The namespace queue on the broker to which the router should route messages. All addresses that start with **queue** will be routed to a queue on the broker.

- 3 The incoming autolink from **queue.first** on the broker to the router.
- 4 The outgoing autolink from the router to **queue.first** on the broker.
- 5 The incoming autolink from **queue.second** on the broker to the router.
- 6 The outgoing autolink from the router to **queue.second** on the broker.

### 8.2.6.2. How the Messages are Routed

Initially, when the broker is offline, the autolinks are inactive.

```
$ qdstat --autolinks
AutoLinks
  addr          dir  phs  extAddr  link  status  lastErr
=====
queue.first    in   1           inactive
queue.first    out  0           inactive
queue.second   in   1           inactive
queue.second   out  0           inactive
```

Once the broker is online, the autolinks attempt to activate. In this case, the broker starts with the **queue.first** queue only, and the **queue.first** autolinks become active. The **queue.second** autolinks are in a failed state, because the **queue.second** queue does not exist on the broker.

```
$ qdstat --autolinks
AutoLinks
  addr          dir  phs  extAddr  link  status  lastErr
=====
=
queue.first    in   1           6    active
queue.first    out  0           7    active
queue.second   in   1           failed  Node not found:
queue.second
queue.second   out  0           failed  Node not found:
queue.second
```

The producer now connects to the router and sends three messages to **queue.first**.

```
$ python simple_send.py -a 127.0.0.1/queue.first -m3
all messages confirmed
```

The router's address statistics show that the messages were delivered to the queue.

```
$ qdstat -a
Router Addresses
  class  addr          phs  distrib  in-proc  local  remote  cntnr  in
out thru to-proc from-proc
```

```

=====
=====
mobile queue.first 1 balanced 0 0 0 0 0
0 0 0 0
mobile queue.first 0 balanced 0 1 0 0 3
3 0 0 0

```

The **queue.first** address appears twice in the output: once for each phase of the address. Phase 0 is for routing messages from producers to the outgoing autolink. Phase 1 is for routing messages from the incoming autolink to the subscribed consumers. In this case, Phase 0 of the address has counted three messages in the **in** column (the messages that arrived on the router from the producer), and three messages in the **out** column (the messages that were sent from the router to the broker queue).

The consumer now connects to the router and receives the three messages from **queue.first**.

```

$ python simple_recv.py -a 127.0.0.1:5672/queue.first -m3
{u'sequence': int32(1)}
{u'sequence': int32(2)}
{u'sequence': int32(3)}

```

The router's address statistics now show that all three messages were received by the consumer from the broker queue.

```

$ qdstat -a
Router Addresses
class addr phs distrib in-proc local remote cntnr in
out thru to-proc from-proc
=====
=====
mobile queue.first 1 balanced 0 0 0 0 3
3 0 0 0
mobile queue.first 0 balanced 0 1 0 0 3
3 0 0 0

```

The command output shows that Phase 1 of the address was used to deliver all three messages from the queue to the consumer.



## NOTE

Even in a multi-router network, and with multiple producers and consumers for **queue.first**, all deliveries are routed through the queue on the connected broker.

## 8.3. CONFIGURING LINK ROUTING

Link routing provides an alternative strategy for brokered messaging. A link route represents a private messaging path between a sender and a receiver in which the router passes the messages between end points. You can think of a link route as a "virtual connection" or "tunnel" that travels from a sender, through the router network, to a receiver.

With link routing, routing is performed on link-attach frames, which are chained together to form a virtual messaging path that directly connects a sender and receiver. Once a link route is established, the transfer of message deliveries, flow frames, and dispositions is performed across the link route.

### 8.3.1. Link Route Addresses

A link route address represents a broker queue, topic, or other service. When a client attaches a link route address to a router, the router propagates a link attachment to the broker resource identified by the address.

### 8.3.2. Link Route Routing Patterns

Routing patterns are not used with link routing, because there is a direct link between the sender and receiver. The router only makes a routing decision when it receives the initial link-attach request frame. Once the link is established, the router passes the messages along the link in a balanced distribution.

### 8.3.3. Link Route Flow Control

Unlike message routing, with link routing, the sender and receiver handle flow control directly: the receiver grants link credits, which is the number of messages it is able to receive. The router sends them directly to the sender, and then the sender sends the messages based on the credits that the receiver granted.

### 8.3.4. Creating a Link Route

Link routes establish a link between a sender and a receiver that travels through a router. You can configure inward and outward link routes to enable the router to receive link-attaches from clients and to send them to a particular destination.

With link routing, client traffic is handled on the broker, not the router. Clients have a direct link through the router to a broker's queue. Therefore, each client is a separate producer or consumer.



#### NOTE

If the connection to the broker fails, the routed links are detached, and the router will attempt to reconnect to the broker (or its backup). Once the connection is reestablished, the link route to the broker will become reachable again.

From the client's perspective, the client will see the detached links (that is, the senders or receivers), but not the failed connection. Therefore, if you want the client to reattach dropped links in the event of a broker connection failure, you must configure this functionality on the client. Alternatively, you can use message routing with autolinks instead of link routing. For more information, see [Routing Messages through a Broker Queue](#).

#### Procedure

1. In the router configuration file, add an outgoing connection to the broker:

```
connector {
  name: NAME
  host: HOST_NAME/ADDRESS
  port: PORT_NUMBER/NAME
  role: route-container
  ...
}
```

**name**

The name of the **connector**. You should specify a name that describes the broker.

#### host

Either an IP address (IPv4 or IPv6) or hostname on which the router should connect to the broker.

#### port

The port number or symbolic service name on which the router should connect to the broker.

#### role

Specify **route-container** to indicate that this connection is for an external container (broker).

For information about additional attributes, see [connector](#) in the **qdrouterd.conf** man page.

- If you want clients to send local transactions to the broker, create a link route for the transaction coordinator:

```
linkRoute {
  prefix: $coordinator 1
  connection: CONNECTOR_NAME
  direction: in
}
```

- The **\$coordinator** prefix designates this link route as a transaction coordinator. When the client opens a transacted session, the requests to start and end the transaction are propagated along this link route to the broker.

AMQ Interconnect does not support routing transactions to multiple brokers. If you have multiple brokers in your environment, choose a single broker and route all transactions to it.

- If you want clients to send messages on this link route, create an incoming link route:

```
linkRoute {
  prefix: ADDRESS_PREFIX
  connection: CONNECTOR_NAME
  direction: in
  ...
}
```

#### prefix | pattern

The address prefix or pattern that matches the broker queue that should be the destination for routed link-attaches. All messages that match this prefix or pattern will be distributed along the link route. You can specify a prefix to match an exact address or beginning segment of an address. Alternatively, you can specify a pattern to match an address using wildcards.

A *prefix* matches either an exact address or the beginning segment within an address that is delimited by either a `.` or `/` character. For example, the prefix **my\_address** would match the address **my\_address** as well as **my\_address.1** and **my\_address/1**. However, it would not match **my\_address1**.

A *pattern* matches an address that corresponds to a pattern. A pattern is a sequence of words delimited by either a `.` or `/` character. You can use wildcard characters to represent a word. The `*` character matches exactly one word, and the `#` character matches any sequence of zero or more words.

The `*` and `#` characters are reserved as wildcards. Therefore, you should not use them in the message address.

For more information about creating address patterns, see [Section 5.3.1, “Pattern Matching for Addresses”](#).



## NOTE

You can convert a **prefix** value to a **pattern** by appending `/#` to it. For example, the prefix `a/b/c` is equivalent to the pattern `a/b/c/#`.

### connection | containerID

How the router should connect to the broker. You can specify either an outgoing connection (**connection**) or the container ID of the broker (**containerID**).

If multiple brokers are connected to the router through this connection, requests for addresses matching the link route’s prefix or pattern are balanced across the brokers.

Alternatively, if you want to specify a particular broker, use **containerID** and add the broker’s container ID.

### direction

Set this attribute to `in` to specify that clients can send messages into the router network on this link route.

For information about additional attributes, see [linkRoute](#) in the `qdrouterd.conf` man page.

- If you want clients to receive messages on this link route, create an outgoing link route:

```
linkRoute {
    prefix: ADDRESS_PREFIX
    connection: CONNECTOR_NAME
    direction: out
    ...
}
```

### prefix | pattern

The address prefix or pattern that matches the broker queue from which you want to receive routed link-attaches. All messages that match this prefix or pattern will be distributed along the link route. You can specify a prefix to match an exact address or beginning segment of an address. Alternatively, you can specify a pattern to match an address using wildcards.

A *prefix* matches either an exact address or the beginning segment within an address that is delimited by either a `.` or `/` character. For example, the prefix `my_address` would match the address `my_address` as well as `my_address.1` and `my_address/1`. However, it would not match `my_address1`.

A *pattern* matches an address that corresponds to a pattern. A pattern is a sequence of words delimited by either a `.` or `/` character. You can use wildcard characters to represent a word.

The `*` character matches exactly one word, and the `#` character matches any sequence of zero or more words.

The `*` and `#` characters are reserved as wildcards. Therefore, you should not use them in the message address.

For more information about creating address patterns, see [Section 5.3.1, “Pattern Matching for Addresses”](#).



## NOTE

You can convert a **prefix** value to a **pattern** by appending `/#` to it. For example, the prefix `a/b/c` is equivalent to the pattern `a/b/c/#`.

### connection | containerID

How the router should connect to the broker. You can specify either an outgoing connection (**connection**) or the container ID of the broker (**containerID**).

If multiple brokers are connected to the router through this connection, requests for addresses matching the link route’s prefix or pattern are balanced across the brokers. Alternatively, if you want to specify a particular broker, use **containerID** and add the broker’s container ID.

### direction

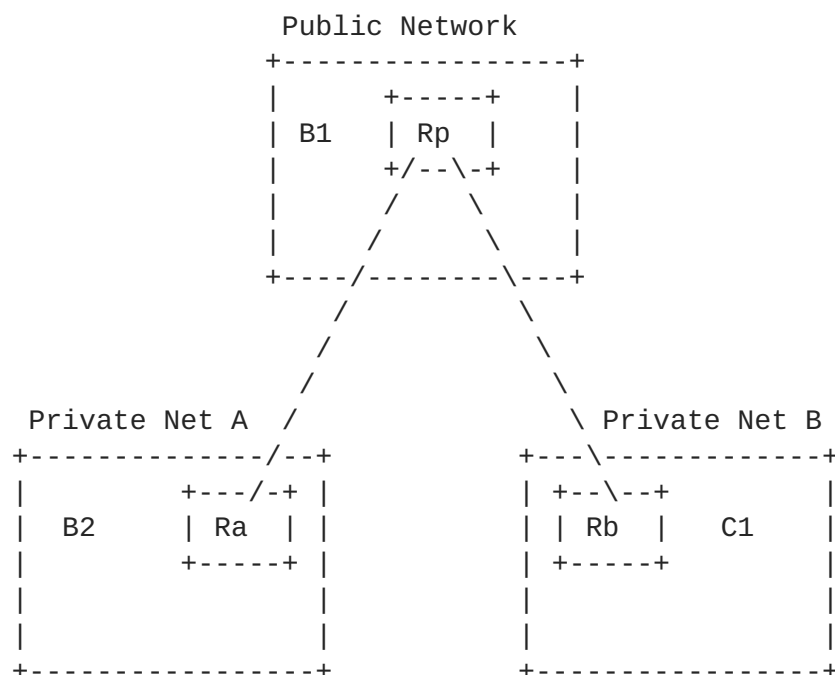
Set this attribute to **out** to specify that this link route is for receivers.

For information about additional attributes, see [linkRoute](#) in the `qdrouterd.conf` man page.

## 8.3.5. Example: Using a Link Route to Provide Client Isolation

This example shows how a link route can connect a client to a message broker that is on a different private network.

### Router Network with Isolated Clients



Client **C1** is constrained by firewall policy to connect to the router in its own network (**Rb**). However, it can use a link route to access queues, topics, and any other AMQP services that are provided on message brokers **B1** and **B2** — even though they are on different networks.



In this example, client **C1** needs to receive messages from **b2.event-queue**, which is hosted on broker **B2** in **Private Net A**. A link route connects the client and broker even though neither of them is aware that there is a router network between them.

### 8.3.5.1. Router Configuration

To enable client **C1** to receive messages from **b2.event-queue** on broker **B2**, router **Ra** must be able to do the following:

- Connect to broker **B2**
- Route links to and from broker **B2**
- Advertise itself to the router network as a valid destination for links that have a **b2.event-queue** address.

The relevant part of the configuration file for router **Ra** shows the following:

```
connector { ❶
  name: broker
  role: route-container
  host: 198.51.100.1
  port: 61617
  saslMechanisms: ANONYMOUS
}

linkRoute { ❷
  prefix: b2
  direction: in
  connection: broker
}

linkRoute { ❸
  prefix: b2
  direction: out
  connection: broker
}
```

- ❶ The outgoing connection from the router to broker **B2**. The **route-container** role enables the router to connect to an external AMQP container (in this case, a broker).
- ❷ The incoming link route for receiving links from client senders. Any sender with a target whose address begins with **b2** will be routed to broker **B2** using the **broker** connector.
- ❸ The outgoing link route for sending links to client receivers. Any receivers whose source address begins with **b2** will be routed to broker **B2** using the **broker** connector.

This configuration enables router **Ra** to advertise itself as a valid destination for targets and sources starting with **b2**. It also enables the router to connect to broker **B2**, and to route links to and from queues starting with the **b2** prefix.

**NOTE**

While not required, routers **Rp** and **Rb** should also have the same configuration.

**8.3.5.2. How the Client Receives Messages**

By using the configured link route, client **C1** can receive messages from broker **B2** even though they are on different networks.

Router **Ra** establishes a connection to broker **B2**. Once the connection is open, **Ra** tells the other routers (**Rp** and **Rb**) that it is a valid destination for link routes to the **b2** prefix. This means that sender and receiver links attached to **Rb** or **Rp** will be routed along the shortest path to **Ra**, which then routes them to broker **B2**.

To receive messages from the **b2.event-queue** on broker **B2**, client **C1** attaches a receiver link with a source address of **b2.event-queue** to its local router, **Rb**. Because the address matches the **b2** prefix, **Rb** routes the link to **Rp**, which is the next hop in the route to its destination. **Rp** routes the link to **Ra**, which routes it to broker **B2**. Client **C1** now has a receiver established, and it can begin receiving messages.

**NOTE**

If broker **B2** is unavailable for any reason, router **Ra** will not advertise itself as a destination for **b2** addresses. In this case, routers **Rb** and **Rp** will reject link attaches that should be routed to broker **B2** with an error message indicating that there is no route available to the destination.

## CHAPTER 9. LOGGING

Logging enables you to diagnose error and performance issues with AMQ Interconnect.

AMQ Interconnect consists of internal modules that provide important information about the router. For each module, you can specify logging levels, the format of the log file, and the location to which the logs should be written.

### 9.1. LOGGING MODULES

AMQ Interconnect logs are broken into different categories called *logging modules*. Each module provides important information about a particular aspect of AMQ Interconnect.

#### 9.1.1. The DEFAULT Logging Module

The default module. This module applies defaults to all of the other logging modules.

#### 9.1.2. The ROUTER Logging Module

This module provides information and statistics about the local router. This includes how the router connects to other routers in the network, and information about the remote destinations that are directly reachable from the router (link routes, waypoints, autolinks, and so on).

In this example, on **Router .A**, the **ROUTER** log shows that **Router .B** is the next hop. It also shows the cost for **Router .A** to reach the other routers on the network:

```
Tue Jun 7 13:28:27 2016 ROUTER (trace) Node Router.C next hop set:
Router.B
Tue Jun 7 13:28:27 2016 ROUTER (trace) Node Router.C valid origins: []
Tue Jun 7 13:28:27 2016 ROUTER (trace) Node Router.C cost: 2
Tue Jun 7 13:28:27 2016 ROUTER (trace) Node Router.B valid origins: []
Tue Jun 7 13:28:27 2016 ROUTER (trace) Node Router.B cost: 1
```

On **Router .B**, the **ROUTER** log provides more information about valid origins:

```
Tue Jun 7 13:28:25 2016 ROUTER (trace) Node Router.C cost: 1
Tue Jun 7 13:28:26 2016 ROUTER (trace) Node Router.A created: maskbit=2
Tue Jun 7 13:28:26 2016 ROUTER (trace) Node Router.A link set: link_id=1
Tue Jun 7 13:28:26 2016 ROUTER (trace) Node Router.A valid origins:
['Router.C']
Tue Jun 7 13:28:26 2016 ROUTER (trace) Node Router.A cost: 1
Tue Jun 7 13:28:27 2016 ROUTER (trace) Node Router.C valid origins:
['Router.A']
```

#### 9.1.3. The ROUTER\_CORE Logging Module

This module provides information about the local router's operations on active connections and links. This includes operations related to opened and closed connections, messages sent, deliveries, and flow control.

```
Tue Jun 7 13:42:07 2016 ROUTER_CORE (trace) Core action 'link_flow'
Tue Jun 7 13:42:08 2016 ROUTER_CORE (trace) Core action 'link_deliver'
```

```
Tue Jun 7 13:42:08 2016 ROUTER_CORE (trace) Core action 'send_to'
Tue Jun 7 13:42:08 2016 ROUTER_CORE (trace) Core action 'link_flow'
```

#### 9.1.4. The ROUTER\_HELLO Logging Module

This module provides information about the *Hello* protocol used by interior routers to exchange Hello messages, which include information about the router's ID and a list of its reachable neighbors (the other routers with which this router has bidirectional connectivity).

The logs for this module are helpful for monitoring or resolving issues in the network topology, and for determining to which other routers a router is connected, and the hop-cost for each of those connections.

In this example, on **Router .A**, the **ROUTER\_HELLO** log shows that it is connected to **Router .B**, and that **Router .B** is connected to **Router .A** and **Router .C**:

```
Tue Jun 7 13:50:21 2016 ROUTER_HELLO (trace) RCVD: HELLO(id=Router.B
area=0 inst=1465307413 seen=['Router.A', 'Router.C']) ❶
Tue Jun 7 13:50:21 2016 ROUTER_HELLO (trace) SENT: HELLO(id=Router.A
area=0 inst=1465307416 seen=['Router.B']) ❷
Tue Jun 7 13:50:22 2016 ROUTER_HELLO (trace) RCVD: HELLO(id=Router.B
area=0 inst=1465307413 seen=['Router.A', 'Router.C'])
Tue Jun 7 13:50:22 2016 ROUTER_HELLO (trace) SENT: HELLO(id=Router.A
area=0 inst=1465307416 seen=['Router.B'])
```

- ❶ **Router .A** received a Hello message from **Router .B**, which can see **Router .A** and **Router .C**.
- ❷ **Router .A** sent a Hello message to **Router .B**, which is the only router it can see.

On **Router .B**, the **ROUTER\_HELLO** log shows the same router topology from a different perspective:

```
Tue Jun 7 13:50:18 2016 ROUTER_HELLO (trace) SENT: HELLO(id=Router.B
area=0 inst=1465307413 seen=['Router.A', 'Router.C']) ❶
Tue Jun 7 13:50:18 2016 ROUTER_HELLO (trace) RCVD: HELLO(id=Router.A
area=0 inst=1465307416 seen=['Router.B']) ❷
Tue Jun 7 13:50:19 2016 ROUTER_HELLO (trace) RCVD: HELLO(id=Router.C
area=0 inst=1465307411 seen=['Router.B']) ❸
```

- ❶ **Router .B** sent a Hello message to **Router .A** and **Router .C**.
- ❷ **Router .B** received a Hello message from **Router .A**, which can only see **Router .B**.
- ❸ **Router .B** received a Hello message from **Router .C**, which can only see **Router .B**.

#### 9.1.5. The ROUTER\_LS Logging Module

This module provides information about link-state data between routers, including Router Advertisement (RA), Link State Request (LSR), and Link State Update (LSU) messages.

Periodically, each router sends an LSR to the other routers and receives an LSU with the requested information. Exchanging the above information, each router can compute the next hops in the topology, and the related costs.

This example shows the RA, LSR, and LSU messages sent between three routers:

```
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) SENT: LSR(id=Router.A area=0)
to: Router.C //
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) SENT: LSR(id=Router.A area=0)
to: Router.B //
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) SENT: RA(id=Router.A area=0
inst=1465308600 ls_seq=1 mobile_seq=1) ❶
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) RCVD: LSU(id=Router.B area=0
inst=1465308595 ls_seq=2 ls=LS(id=Router.B area=0 ls_seq=2 peers=
{'Router.A': 1L, 'Router.C': 1L})) ❷
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) RCVD: LSR(id=Router.B area=0)
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) SENT: LSU(id=Router.A area=0
inst=1465308600 ls_seq=1 ls=LS(id=Router.A area=0 ls_seq=1 peers=
{'Router.B': 1}))
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) RCVD: RA(id=Router.C area=0
inst=1465308592 ls_seq=1 mobile_seq=0)
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) SENT: LSR(id=Router.A area=0)
to: Router.C
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) RCVD: LSR(id=Router.C area=0)
❸
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) SENT: LSU(id=Router.A area=0 //
inst=1465308600 ls_seq=1 ls=LS(id=Router.A area=0 ls_seq=1 peers=
{'Router.B': 1}))
Tue Jun  7 14:10:02 2016 ROUTER_LS (trace) RCVD: LSU(id=Router.C area=0
inst=1465308592 ls_seq=1 ls=LS(id=Router.C area=0 ls_seq=1 peers=
{'Router.B': 1L})) ❹
Tue Jun  7 14:10:03 2016 ROUTER_LS (trace) Computed next hops:
{'Router.C': 'Router.B', 'Router.B': 'Router.B'} ❺
Tue Jun  7 14:10:03 2016 ROUTER_LS (trace) Computed costs: {'Router.C':
2L, 'Router.B': 1}
Tue Jun  7 14:10:03 2016 ROUTER_LS (trace) Computed valid origins:
{'Router.C': [], 'Router.B': []}
```

- ❶ **Router .A** sent LSR requests and an RA advertisement to the other routers on the network.
- ❷ **Router .A** received an LSU from **Router .B**, which has two peers: **Router .A**, and **Router .C** (with a cost of 1).
- ❸ **Router .A** received an LSR from both **Router .B** and **Router .C**, and replied with an LSU.
- ❹ **Router .A** received an LSU from **Router .C**, which only has one peer: **Router .B** (with a cost of 1).
- ❺ After the LSR and LSU messages are exchanged, **Router .A** computed the router topology with the related costs.

### 9.1.6. The ROUTER\_MA Logging Module

This module provides information about the exchange of mobile address information between routers, including Mobile Address Request (MAR) and Mobile Address Update (MAU) messages exchanged between routers. You can use this log to monitor the state of mobile addresses attached to each router.

This example shows the MAR and MAU messages sent between three routers:

```
Tue Jun 7 14:27:20 2016 ROUTER_MA (trace) SENT: MAU(id=Router.A area=0
mobile_seq=1 add=['Cmy_queue', 'Dmy_queue', 'M0my_queue_wp'] del=[]) ❶
Tue Jun 7 14:27:21 2016 ROUTER_MA (trace) RCVD: MAR(id=Router.C area=0
have_seq=0) ❷
Tue Jun 7 14:27:21 2016 ROUTER_MA (trace) SENT: MAU(id=Router.A area=0
mobile_seq=1 add=['Cmy_queue', 'Dmy_queue', 'M0my_queue_wp'] del=[])
Tue Jun 7 14:27:22 2016 ROUTER_MA (trace) RCVD: MAR(id=Router.B area=0
have_seq=0) ❸
Tue Jun 7 14:27:22 2016 ROUTER_MA (trace) SENT: MAU(id=Router.A area=0
mobile_seq=1 add=['Cmy_queue', 'Dmy_queue', 'M0my_queue_wp'] del=[])
Tue Jun 7 14:27:39 2016 ROUTER_MA (trace) RCVD: MAU(id=Router.C area=0
mobile_seq=1 add=['M0my_test'] del=[]) ❹
Tue Jun 7 14:27:51 2016 ROUTER_MA (trace) RCVD: MAU(id=Router.C area=0
mobile_seq=2 add=[] del=['M0my_test']) ❺
```

- ❶ **Router .A** sent MAU messages to the other routers in the network to notify them about the addresses added for **my\_queue** and **my\_queue\_wp**.
- ❷ **Router .A** received a MAR message in response from **Router .C**.
- ❸ **Router .A** received another MAR message in response from **Router .B**.
- ❹ **Router .C** sent a MAU message to notify the other routers that it added and address for **my\_test**.
- ❺ **Router .C** sent another MAU message to notify the other routers that it deleted the address for **my\_test** (because the receiver is detached).

### 9.1.7. The MESSAGE Logging Module

This module provides information about AMQP messages sent and received by the router, including information about the address, body, and link. You can use this log to find high-level information about messages on a particular router.

In this example, **Router .A** has sent and received some messages related to the Hello protocol, and sent and received some other messages on a link for a mobile address:

```
Tue Jun 7 14:36:54 2016 MESSAGE (trace) Sending
Message{to='amqp://_topo/0/Router.B/qdrouter'
body='\d1\00\00\00\1b\00\00\00\04\01\02id\01\08R'} on link
qdlink.p9XmBm19uDqx50R
Tue Jun 7 14:36:54 2016 MESSAGE (trace) Received
Message{to='amqp://_topo/0/Router.A/qdrouter'
body='\d1\00\00\00\8e\00\00\00
\01\06ls_se'} on link qdlink.phMsJ0q7YaFsGAG
Tue Jun 7 14:36:54 2016 MESSAGE (trace) Received Message{
body='\d1\00\00\00\10\00\00\00\02\01\08seque'} on link
qdlink.FYHqBX+TtwXZHfV
Tue Jun 7 14:36:54 2016 MESSAGE (trace) Sending Message{
body='\d1\00\00\00\10\00\00\00\02\01\08seque'} on link
qdlink.yU1tnPs5KbMlieM
Tue Jun 7 14:36:54 2016 MESSAGE (trace) Sending
```

```

Message{to='amqp:/_local/qdhello'
body='\d1\00\00\00G\00\00\00\00\08\a1\04seen\d0'} on link
qdlink.p9XmBm19uDqx50R
Tue Jun 7 14:36:54 2016 MESSAGE (trace) Sending
Message{to='amqp:/_topo/0/Router.C/qdrouter'
body='\d1\00\00\00\1b\00\00\00\04\a1\02id\a1\08R'} on link
qdlink.p9XmBm19uDqx50R

```

### 9.1.8. The SERVER Logging Module

This module provides information about how the router is listening for and connecting to other containers in the network (such as clients, routers, and brokers). This includes the state of AMQP messages sent and received by the broker (open, begin, attach, transfer, flow, and so on), and the related content of those messages.

For example, this log shows details about how the router handled a link attachment:

```

Tue Jun 7 14:39:52 2016 SERVER (trace) [2]: <- AMQP
Tue Jun 7 14:39:52 2016 SERVER (trace) [1]: <- AMQP
Tue Jun 7 14:39:52 2016 SERVER (trace) [1]:0 <- @open(16) [container-
id="Router.B", max-frame-size=16384, channel-max=32767, idle-time-
out=8000, offered-capabilities="ANONYMOUS-RELAY", properties=
{:product="qpidd-dispatch-router", :version="0.6.0"}]
Tue Jun 7 14:39:52 2016 SERVER (trace) [1]:0 -> @begin(17) [next-
outgoing-id=0, incoming-window=15, outgoing-window=2147483647]
Tue Jun 7 14:39:52 2016 SERVER (trace) [1]:RAW:
"\x00\x00\x00\x1e\x02\x00\x00\x00\x00S\x11\xd0\x00\x00\x00\x0e\x00\x00\x00
\x04@R\x00R\x0fp\x7f\xff\xff\xff"
Tue Jun 7 14:39:52 2016 SERVER (trace) [1]:1 -> @begin(17) [next-
outgoing-id=0, incoming-window=15, outgoing-window=2147483647]
Tue Jun 7 14:39:52 2016 SERVER (trace) [1]:RAW:
"\x00\x00\x00\x1e\x02\x00\x00\x01\x00S\x11\xd0\x00\x00\x00\x0e\x00\x00\x00
\x04@R\x00R\x0fp\x7f\xff\xff\xff"
Tue Jun 7 14:39:52 2016 SERVER (trace) [1]:0 -> @attach(18)
[name="qdlink.uSseXPSfTHhxo8d", handle=0, role=true, snd-settle-mode=2,
rcv-settle-mode=0, source=@source(40) [durable=0, expiry-policy="link-
detach", timeout=0, dynamic=false, capabilities="qd.router"],
target=@target(41) [durable=0, expiry-policy="link-detach", timeout=0,
dynamic=false, capabilities="qd.router"], initial-delivery-count=0]
Tue Jun 7 14:39:52 2016 SERVER (trace) [1]:RAW:
"\x00\x00\x00\x91\x02\x00\x00\x00\x00S\x12\xd0\x00\x00\x00\x81\x00\x00\x00
\x0a\xa1\x16qdlink.uSseXPSfTHhxo8dR\x00AP\x02P\x00\x00S(\xd0\x00\x00\x00'\
\x00\x00\x00\x0b@R\x00\xa3\x0blink-
detachR\x00B@@@@@xa3\x09qd.router\x00S)\xd0\x00\x00\x00#\x00\x00\x00\x07@
R\x00\xa3\x0blink-detachR\x00B@xa3\x09qd.router@@R\x00"

```

### 9.1.9. The AGENT Logging Module

This module provides information about configuration changes made to the router from either editing the router's configuration file or using **qdmange**.

In this example, on **Router.A**, **address**, **linkRoute**, and **autoLink** entities were added to the router's configuration file. When the router was started, the **AGENT** module applied these changes, and they are now viewable in the log:

```

Tue Jun  7 15:07:32 2016 AGENT (debug) Add entity:
ConnectorEntity(addr=127.0.0.1, allowRedirect=True, cost=1,
host=127.0.0.1, identity=connector/127.0.0.1:5672:BROKER,
idleTimeoutSeconds=16, maxFrameSize=65536, name=BROKER, port=5672,
role=route-container, stripAnnotations=both,
type=org.apache.qpid.dispatch.connector, verifyHostname=True)
Tue Jun  7 15:07:32 2016 AGENT (debug) Add entity:
RouterConfigAddressEntity(distribution=closest,
identity=router.config.address/0, name=router.config.address/0,
prefix=my_address, type=org.apache.qpid.dispatch.router.config.address,
waypoint=False)
Tue Jun  7 15:07:32 2016 AGENT (debug) Add entity:
RouterConfigAddressEntity(distribution=balanced,
identity=router.config.address/1, name=router.config.address/1,
prefix=my_queue_wp, type=org.apache.qpid.dispatch.router.config.address,
waypoint=True)
Tue Jun  7 15:07:32 2016 AGENT (debug) Add entity:
RouterConfigLinkrouteEntity(connection=BROKER, direction=in,
distribution=linkBalanced, identity=router.config.linkRoute/0,
name=router.config.linkRoute/0, prefix=my_queue,
type=org.apache.qpid.dispatch.router.config.linkRoute)
Tue Jun  7 15:07:32 2016 AGENT (debug) Add entity:
RouterConfigLinkrouteEntity(connection=BROKER, direction=out,
distribution=linkBalanced, identity=router.config.linkRoute/1,
name=router.config.linkRoute/1, prefix=my_queue,
type=org.apache.qpid.dispatch.router.config.linkRoute)
Tue Jun  7 15:07:32 2016 AGENT (debug) Add entity:
RouterConfigAutolinkEntity(addr=my_queue_wp, connection=BROKER,
direction=in, identity=router.config.autoLink/0,
name=router.config.autoLink/0,
type=org.apache.qpid.dispatch.router.config.autoLink)
Tue Jun  7 15:07:32 2016 AGENT (debug) Add entity:
RouterConfigAutolinkEntity(addr=my_queue_wp, connection=BROKER,
direction=out, identity=router.config.autoLink/1,
name=router.config.autoLink/1,
type=org.apache.qpid.dispatch.router.config.autoLink)

```

### 9.1.10. The CONTAINER Logging Module

This module provides information about the nodes related to the router. This includes only the AMQP relay node.

```

Tue Jun  7 14:46:18 2016 CONTAINER (trace) Container Initialized
Tue Jun  7 14:46:18 2016 CONTAINER (trace) Node Type Registered - router
Tue Jun  7 14:46:18 2016 CONTAINER (trace) Node of type 'router' installed
as default node

```

### 9.1.11. The ERROR Logging Module

This module provides detailed information about error conditions encountered during execution.

In this example, **Router.A** failed to start when an incorrect path was specified for the router's configuration file:

■



```

$ sudo qdrouterd --conf xxx
Wed Jun 15 09:53:28 2016 ERROR (error) Python: Exception: Cannot load
configuration file xxx: [Errno 2] No such file or directory: 'xxx'
Wed Jun 15 09:53:28 2016 ERROR (error) Traceback (most recent call last):
  File "/usr/lib/qpid-
dispatch/python/qpid_dispatch_internal/management/config.py", line 155, in
configure_dispatch
    config = Config(filename)
  File "/usr/lib/qpid-
dispatch/python/qpid_dispatch_internal/management/config.py", line 41, in
__init__
    self.load(filename, raw_json)
  File "/usr/lib/qpid-
dispatch/python/qpid_dispatch_internal/management/config.py", line 123, in
load
    with open(source) as f:
Exception: Cannot load configuration file xxx: [Errno 2] No such file or
directory: 'xxx'

Wed Jun 15 09:53:28 2016 MAIN (critical) Router start-up failed: Python:
Exception: Cannot load configuration file xxx: [Errno 2] No such file or
directory: 'xxx'
qdrouterd: Python: Exception: Cannot load configuration file xxx: [Errno
2] No such file or directory: 'xxx'

```

### 9.1.12. The POLICY Logging Module

This module provides information about policies that have been configured for the router.

In this example, **Router.A** has no limits on maximum connections, and the default application policy is disabled:

```

Tue Jun  7 15:07:32 2016 POLICY (info) Policy configured
maximumConnections: 0, policyFolder: '', access rules enabled: 'false'
Tue Jun  7 15:07:32 2016 POLICY (info) Policy fallback defaultApplication
is disabled

```

## 9.2. CONFIGURING LOGGING

You can specify the types of events that should be logged, the format of the log entries, and where those entries should be sent.

### Procedure

1. In the router's configuration file, add a **log** section to set the default logging properties:

```

log {
    module: DEFAULT
    enable: LOGGING_LEVEL
    includeTimestamp: yes
    ...
}

```

**module**

Specify **DEFAULT**.

### **enable**

The logging level. You can specify any of the following levels (from lowest to highest):

- **trace** - provides the most information, but significantly affects system performance
- **debug** - useful for debugging, but affects system performance
- **info** - provides general information without affecting system performance
- **notice** - provides general information, but is less verbose than **info**
- **warning** - provides information about issues you should be aware of, but which are not errors
- **error** - error conditions that you should address
- **critical** - critical system issues that you must address immediately

To specify multiple levels, use a comma-separated list. You can also use **+** to specify a level and all levels above it. For example, **trace, debug, warning+** enables trace, debug, warning, error, and critical levels. For default logging, you should typically use the **info+** or **notice+** level. These levels will provide general information, warnings, and errors for all modules without affecting the performance of AMQ Interconnect.

### **includeTimestamp**

Set this to **yes** to include the timestamp in all logs.

For information about additional log attributes, see [log](#) in the **qdrouterd.conf** man page.

2. Add an additional **log** section for each logging module that should not follow the default logging configuration:

```
log {
    module: MODULE_NAME
    enable: LOGGING_LEVEL
    ...
}
```

### **module**

The name of the module for which you are configuring logging. For a list of valid modules, see [Section 9.1, “Logging Modules”](#).

### **enable**

The logging level. You can specify any of the following levels (from lowest to highest):

- **trace** - provides the most information, but significantly affects system performance
- **debug** - useful for debugging, but affects system performance
- **info** - provides general information without affecting system performance
- **notice** - provides general information, but is less verbose than **info**

- **warning** - provides information about issues you should be aware of, but which are not errors
- **error** - error conditions that you should address
- **critical** - critical system issues that you must address immediately

To specify multiple levels, use a comma-separated list. You can also use **+** to specify a level and all levels above it. For example, **trace, debug, warning+** enables trace, debug, warning, error, and critical levels. For default logging, you should typically use the **info+** or **notice+** level. These levels will provide general information, warnings, and errors for all modules without affecting the performance of AMQ Interconnect.

For information about additional log attributes, see [log](#) in the **qdrouterd.conf** man page.

## 9.3. VIEWING LOG ENTRIES

You may need to view log entries to diagnose errors, performance problems, and other important issues. A log entry consists of an optional timestamp, the logging module, the logging level, and the log message.

### 9.3.1. Viewing Log Entries on the Console

By default, log entries are logged to the console, and you can view them there. However, if the **output** attribute is set for a particular logging module, then you can find those log entries in the specified location (**stderr**, **syslog**, or a file).

### 9.3.2. Viewing Log Entries on the CLI

You can use the **qdstat** tool to view a list of recent log entries.

#### Procedure

- Use the **qdstat --log** command to view recent log entries.  
You can use the **--limit** parameter to limit the number of log entries that are displayed. For more information about **qdstat**, see [qdstat man page](#).

This example displays the last three log entries for **Router.A**:

```
$ qdstat --log --limit=3 -r ROUTER.A
Wed Jun  7 17:49:32 2017 ROUTER_CORE (none) Core action
'link_deliver'
Wed Jun  7 17:49:32 2017 ROUTER_CORE (none) Core action 'send_to'
Wed Jun  7 17:49:32 2017 SERVER (none) [2]:0 -> @flow(19) [next-
incoming-id=1, incoming-window=61, next-outgoing-id=0, outgoing-
window=2147483647, handle=0, delivery-count=1, link-credit=250,
drain=false]
```

## CHAPTER 10. MANAGEMENT

You can manage AMQ Interconnect using both graphical and command-line tools.

### AMQ Console

A graphical tool for monitoring and managing AMQ brokers and routers.

### qdstat

A command-line tool for monitoring the status of AMQ Interconnect routers.

### qdmmanage

A command-line tool for viewing and updating the configuration of AMQ Interconnect routers.

## 10.1. USING AMQ CONSOLE

If you prefer to use a graphic interface to manage AMQ, you can use AMQ Console. AMQ Console is a web console included in the AMQ Broker installation, and it enables you to use a web browser to manage AMQ Broker and AMQ Interconnect.

For more information, see [Using AMQ Console](#).

## 10.2. MONITORING AMQ INTERCONNECT USING `qdstat`

You can use `qdstat` to view the status of routers on your router network. For example, you can view information about the attached links and configured addresses, available connections, and nodes in the router network.

### 10.2.1. Syntax for Using `qdstat`

You can use `qdstat` with the following syntax:

```
$ qdstat OPTION [CONNECTION_OPTIONS] [SECURE_CONNECTION_OPTIONS]
```

This specifies:

- An **option** for the type of information to view.
- One or more optional **connection\_options** to specify a router for which to view the information.  
If you do not specify a connection option, `qdstat` connects to the router listening on localhost and the default AMQP port (5672).
- The **secure\_connection\_options** if the router for which you want to view information only accepts secure connections.

For more information about `qdstat`, see the [qdstat man page](#).

### 10.2.2. Viewing General Statistics for a Router

You can view information about a router in the router network, such as its working mode and ID.

#### Procedure

- Use the following command:

```
$ qdstat -g [CONNECTION_OPTIONS]
```

This example shows general statistics for the local router:

```
$ qdstat -g
Router Statistics
attr                                     value
=====
Version                                1.2.0
Mode                                    standalone
Router Id                              Router.A
Link Routes                            0
Auto Links                             0
Links                                  2
Nodes                                  0
Addresses                              4
Connections                            1
Presettled Count                       0
Dropped Presettled Count               0
Accepted Count                         2
Rejected Count                         0
Released Count                        0
Modified Count                        0
Ingress Count                          2
Egress Count                           1
Transit Count                          0
Deliveries from Route Container        0
Deliveries to Route Container          0
```

### 10.2.3. Viewing a List of Connections to a Router

You can view:

- Connections from clients (sender/receiver)
- Connections from and to other routers in the network
- Connections to other containers (such as brokers)
- Connections from the tool itself

#### Procedure

- Use this command:

```
$ qdstat -c [CONNECTION_OPTIONS]
```

For more information about the fields displayed by this command, see [the qdstat -c output columns](#).

In this example, two clients are connected to **Router .A**. **Router .A** is connected to **Router .B** and a broker.

Viewing the connections on Router.A displays the following:

```
$ qdstat -c -r Router.A
Connections
id      host                                container
role    dir  security    authentication    tenant
=====
=====
  2    127.0.0.1:5672
route-container out  no-security    anonymous-user    1
 10    127.0.0.1:5001
inter-router   out  no-security    anonymous-user    2
 12    localhost.localdomain:42972 161211fe-ba9e-4726-9996-
52d6962d1276 normal          in    no-security    anonymous-user
 3
 14    localhost.localdomain:42980 a35fcc78-63d9-4bed-b57c-
053969c38fda normal          in    no-security    anonymous-user
 4
 15    localhost.localdomain:42982 0a03aa5b-7c45-4500-8b38-
db81d01ce651 normal          in    no-security    anonymous-user
 5
```

- 1 This connection shows that **Router.A** is connected to a broker, because the **role** is **route-container**, and the **dir** is **out**.
- 2 **Router.A** is also connected to another router on the network (the **role** is **inter-router**), establishing an output connection (the **dir** is **out**).
- 3 4 These connections show that two clients are connected to **Router.A**, because the **role** is **normal**, and the **dir** is **in**.
- 5 The connection from **qdstat** to **Router.A**. This is the connection that **qdstat** uses to query **Router.A** and display the command output.

**Router.A** is connected to **Router.B**. Viewing the connections on **Router.B** displays the following:

```
$ qdstat -c -r Router.B
Connections
id      host                                container  role    dir
security authentication    tenant
=====
=====
  1    localhost.localdomain:51848 Router.A  inter-router in  no-
security anonymous-user    1
```

- 1 This connection shows that **Router.B** is connected to **Router.A** through an incoming connection (the **role** is **inter-router** and the **dir** is **in**). There is not a connection from **qdstat** to **Router.B**, because the command was run from **Router.A** and forwarded to **Router.B**.

#### 10.2.4. Viewing AMQP Links Attached to a Router

You can view a list of AMQP links attached to the router from clients (sender/receiver), from or to other routers into the network, to other containers (for example, brokers), and from the tool itself.

## Procedure

- Use this command:

```
$ qdstat -l [CONNECTION_OPTIONS]
```

For more information about the fields displayed by this command, see [the qdstat -l output columns](#).

In this example, **Router .A** is connected to both **Router .B** and a broker. A link route is configured for the **my\_queue** queue and waypoint (with autolinks), and for the **my\_queue\_wp** queue on the broker. In addition, there is a receiver connected to **my\_address** (message routing based), another to **my\_queue**, and the a third one to **my\_queue\_wp**.

In this configuration, the router uses only one connection to the broker for both the waypoints (related to **my\_queue\_wp**) and the link route (related to **my\_queue**).

Viewing the links displays the following:

```
$ qdstat -l
Router Links
  type          dir conn id id peer class  addr
  phs cap undel unse del presett psdrop acc rej rel mod
  admin      oper

=====
=====
=====
  router-control in 2      7
250 0 0      2876 0      0      0 0 0 0
enabled up 1
  router-control out 2      8
250 0 0      2716 0      0      0 0 0 0
enabled up
  inter-router in 2      9
250 0 0      1 0      0      0 0 0 0
enabled up
  inter-router out 2      10
250 0 0      1 0      0      0 0 0 0
enabled up
  endpoint in 1      11
1 250 0      0 3 0      0 0 0 0 0
enabled up 2
  endpoint out 1      12
0 250 0      0 3 0      0 0 0 0 0
enabled up
  endpoint out 4      15
0 250 0      0 0 0      0 0 0 0 0
enabled up 3
  endpoint out 6      18 19
250 0 0      1 0      0      0 0 0 0
enabled up 4
```

```

    endpoint      in  1      19 18
0    0    0      1    0      0    0    0    0
enabled up 5
    endpoint      out 19      40      mobile my_queue_wp
1    250 0      0    1    0      0    0    0    0
enabled up 6
    endpoint      in  24      48      mobile $management
0    250 0      0    1    0      0    0    0    0
enabled up
    endpoint      out 24      49      local
temp.mx5HxzUe2Eddw_s      250 0      0    0    0    0
0    0    0    0    enabled up

```

- 1** The **conn id 2** connection has four links (in both directions) for inter-router communications with **Router .B**, such as control messages and normal message-routed deliveries.
- 2** There are two autolinks (**conn id 1**) for the waypoint for **my\_queue\_wp**. There is an incoming (**id 11**) and outgoing (**id 12**) link to the broker, and another **out** link (**id 40**) to the receiver.
- 3** A **mobile** link for **my\_address**. The **dir** is **out** related to the receiver attached to it.
- 4** The **out** link from the router to the receiver for **my\_queue**. This enables the router to deliver messages to the receiver.
- 5** The **in** link to the router for **my\_queue**. This enables the router to get messages from **my\_queue** so that they can be sent to the receiver on the **out** link.
- 6** The remaining links are related to the **\$management** address and are used by **qdstat** to receive the information that is displayed by this command.

### 10.2.5. Viewing Known Routers on a Network

To see the topology of the router network, you can view known routers on the network.

#### Procedure

- Use this command:

```
$ qdstat -n [CONNECTION_OPTIONS]
```

For more information about the fields displayed by this command, see [the qdstat -n output columns](#).

In this example, **Router .A** is connected to **Router .B**, which is connected to **Router .C**. Viewing the router topology on **Router .A** shows the following:

```

$ qdstat -n -r Router.A
Routers in the Network
  router-id  next-hop  link  cost  neighbors  valid-
origins
=====

```



```

=====
Router.A   (self)   -           ['Router.B']           [] 1
Router.B   -         0       1     ['Router.A', 'Router.C'] [] 2
Router.C   Router.B -       2     ['Router.B']           [] 3

```

- 1 **Router .A** has one neighbor: **Router .B**.
- 2 **Router .B** is connected to **Router .A** and **Router .C** over **link 0**. The **cost** for **Router .A** to reach **Router .B** is 1, because the two routers are connected directly.
- 3 **Router .C** is connected to **Router .B**, but not to **Router .A**. The **cost** for **Router .A** to reach **Router .C** is 2, because messages would have to pass through **Router .B** as the **next-hop**.

**Router .B** shows a different view of the router topology:

```

$ qdstat -n -v -r Router.B
Routers in the Network
  router-id  next-hop  link  cost  neighbors                                valid-
origins
=====
=====
Router.A    -         0       1     ['Router.B']
['Router.C']
Router.B    (self)   -           ['Router.A', 'Router.C'] []
Router.C    -         1       1     ['Router.B']
['Router.A']

```

The **neighbors** list is the same when viewed on **Router .B**. However, from the perspective of **Router .B**, the destinations on **Router .A** and **Router .C** both have a **cost** of 1. This is because **Router .B** is connected to **Router .A** and **Router .C** through links.

The **valid-origins** column shows that starting from **Router .C**, **Router .B** has the best path to reach **Router .A**. Likewise, starting from **Router .A**, **Router .B** has the best path to reach **Router .C**.

Finally, **Router .C** shows the following details about the router topology:

```

$ qdstat -n -v -r Router.C
Routers in the Network
  router-id  next-hop  link  cost  neighbors                                valid-
origins
=====
=====
Router.A    Router.B  -       2     ['Router.B']           []
Router.B    -         0       1     ['Router.A', 'Router.C'] []
Router.C    (self)   -           ['Router.B']           []

```

Due to a symmetric topology, the **Router .C** perspective of the topology is very similar to the **Router .A** perspective. The primary difference is the **cost**: the cost to reach **Router .B** is 1, because the two routers are connected. However, the cost to reach **Router .A** is 2, because the

messages would have to pass through **Router .B** as the **next-hop**.

## 10.2.6. Viewing Addresses Known to a Router

You can view message-routed and link-routed addresses known to a router.

### Procedure

- Use the following command:

```
$ qdstat -a [CONNECTION_OPTIONS]
```

For more information about the fields displayed by this command, see [the qdstat -a output columns](#).

In this example, **Router .A** is connected to both **Router .B** and a broker. The broker has two queues:

- **my\_queue** (with a link route on **Router .A**)
- **my\_queue\_wp** (with a waypoint and autolinks configured on **Router .A**)

In addition, there are three receivers: one connected to **my\_address** for message routing, another connected to **my\_queue**, and the last one connected to **my\_queue\_wp**.

Viewing the addresses displays the following information:

```
$ qdstat -a
Router Addresses
  class      addr
  local  remote  cntnr  in  out  thru  phs  distrib  in-proc
                to-proc  from-proc
=====
=====
  local      $_management_internal      closest      1      0
0      0      0  0  0      0      0
  local      $displayname      closest      1      0
0      0      0  0  0      0      0
  mobile      $management      0      closest      1      0
0      0      8  0  0      8      0
  local      $management      closest      1      0
0      0      0  0  0      0      0
  router      Router.B      closest      0      0
1      0      0  0  5      0      5 1
  mobile      my_address      0      closest      0      1
0      0      1  1  0      0      0 2
  link-in      my_queue      linkBalanced  0      0
0      1      0  0  0      0      0 3
  link-out      my_queue      linkBalanced  0      0
0      1      0  0  0      0      0
  mobile      my_queue_wp      1      balanced      0      1
0      0      1  1  0      0      0 4
  mobile      my_queue_wp      0      balanced      0      1
0      0      1  1  0      0      0
  local      qdhello      flood      1      1
```

```

0      0      0  0      0      741      706 5
  local      qdrouter      flood      1      0
0      0      0  0      0      4      0
  topo      qdrouter      flood      1      0
1      0      0  0      27  28      28
  local      qdrouter.ma      multicast      1      0
0      0      0  0      0      1      0
  topo      qdrouter.ma      multicast      1      0
1      0      0  0      2      0      3
  local      temp.IJSoXoY_lX0TiDE      closest      0      1
0      0      0  0      0      0      0

```

- 1 An address related to **Router .B** with a **remote** at 1. This is the consumer from **Router .B**.
- 2 The **my\_address** address has one local consumer, which is related to the single receiver attached on that address. The **in** and **out** fields are both 1, which means that one message has traveled through this address using the **closest** distribution method.
- 3 The incoming link route for the **my\_queue** address. This address has one locally-attached container (**cntnr**) as a destination (in this case, the broker). The following entry is the outgoing link for the same address.
- 4 The incoming autolink for the **my\_queue\_wp** address and configured waypoint. There is one local consumer (**local**) for the attached receiver. The following entry is the outgoing autolink for the same address. A single message has traveled through the autolinks.
- 5 The **qdhello**, **qdrouter**, and **qdrouter.ma** addresses are used to periodically update the network topology and deliver router control messages. These updates are made automatically through the inter-router protocol, and are based on all of the messages the routers have exchanged. In this case, the distribution method (**distrib**) for each address is either flood or multicast to ensure the control messages reach all of the routers in the network.

### 10.2.7. Viewing a Router's Autolinks

You can view a list of the autolinks that are associated with waypoint addresses for a node on another container (such as a broker).

#### Procedure

- Use the following command:

```
$ qdstat --autolinks [CONNECTION_OPTIONS]
```

For more information about the fields displayed by this command, see [the `qdstat --autolinks` output columns](#).

In this example, a router is connected to a broker. The broker has a queue called **my\_queue\_wp**, to which the router is configured with a waypoint and autolinks. Viewing the autolinks displays the following:

```
$ qdstat --autolinks
```

```

AutoLinks
  addr          dir  phs  link  status  lastErr
=====
my_queue_wp    in   1   4   active  1
my_queue_wp    out  0   5   active  2

```

- 1 The incoming autolink from **my\_queue\_wp**. As indicated by the **status** field, the link is active, because the broker is running and the connection for the link is already established (as indicated by the **link** field).
- 2 The outgoing autolink to **my\_queue\_wp**. Like the incoming link, it is active and has an established connection.

### 10.2.8. Viewing the Status of a Router's Link Routes

You can view the status of each incoming and outgoing link route.

#### Procedure

- Use the following command:

```
$ qdstat --linkroutes [CONNECTION_OPTIONS]
```

For more information about the fields displayed by this command, see [the `qdstat --linkroutes` output columns](#).

In this example, a router is connected to a broker. The router is configured with a link route to the **my\_queue** queue on the broker. Viewing the link routes displays the following:

```

$ qdstat --linkroutes
Link Routes
  prefix    dir  distrib      status
=====
my_queue   in   linkBalanced  active  1
my_queue   out  linkBalanced  active  2

```

- 1 The incoming link route from **my\_queue** to the router. This route is currently active, because the broker is running.
- 2 The outgoing link from the router to **my\_queue**. This route is also currently active.

### 10.2.9. Viewing Memory Consumption Information

If you need to perform debugging or tracing for a router, you can view information about its memory consumption.

#### Procedure

- Use the following command:

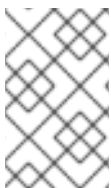
```
$ qdstat -m [CONNECTION_OPTIONS]
```

This command displays information about allocated objects, their size, and their usage by application threads:

```
$ qdstat -m
Types
  type                size  batch  thread-max  total  in-
threads  rebal-in  rebal-out
=====
=====
  qd_bitmask_t         24    64    128         64    64
0          0
  qd_buffer_t          536    16    32         80    80
0          0
  qd_composed_field_t   64    64    128        256   256
0          0
  qd_composite_t       112    64    128        320   320
0          0
  ...
```

### 10.3. MANAGING AMQ INTERCONNECT USING `QDMANAGE`

You can use **qdmmanage** to view and modify the configuration of a running router at runtime. Specifically, **qdmmanage** enables you to create, read, update, and delete the sections and attributes in the router's configuration file without having to restart the router.



#### NOTE

The **qdmmanage** tool implements the AMQP management specification, which means that you can use it with any standard AMQP-managed endpoint, not just with AMQ Interconnect.

#### 10.3.1. Syntax for Using `qdmmanage`

You can use **qdmmanage** with the following syntax:

```
$ qdmmanage [CONNECTION_OPTIONS] OPERATION [OPTIONS]
```

This specifies:

- One or more optional **connection\_options** to specify the router on which to perform the operation, or to supply security credentials if the router only accepts secure connections. If you do not specify any connection options, **qdmmanage** connects to the router listening on localhost and the default AMQP port (5672).
- The **operation** to perform on the router.
- One or more optional **options** to specify a configuration entity on which to perform the operation or how to format the command output.

When you enter a **qdmmanage** command, it is executed as an AMQP management operation request, and then the response is returned as command output in JSON format.

For example, the following command executes a query operation on a router, and then returns the response in JSON format:

```
$ qdmanage query --type listener
[
  {
    "stripAnnotations": "both",
    "addr": "127.0.0.1",
    "multiTenant": false,
    "requireSsl": false,
    "idleTimeoutSeconds": 16,
    "saslMechanisms": "ANONYMOUS",
    "maxFrameSize": 16384,
    "requireEncryption": false,
    "host": "0.0.0.0",
    "cost": 1,
    "role": "normal",
    "http": false,
    "maxSessions": 32768,
    "authenticatePeer": false,
    "type": "org.apache.qpid.dispatch.listener",
    "port": "amqp",
    "identity": "listener/0.0.0.0:amqp",
    "name": "listener/0.0.0.0:amqp"
  }
]
```

For more information about **qdmanage**, see the [qdmanage man page](#).

## 10.3.2. Managing Network Connections

You can use **qdmanage** to view, create, update, and delete listeners and connectors for any router in your router network.

### 10.3.2.1. Managing Listeners

Listeners define how clients can connect to a router. The following table lists the **qdmanage** commands you can use to perform common operations on listeners.

For more information about the attributes you can use with these commands, see [listener](#) in the **qdrouterd.conf** man page.



#### NOTE

The commands in this table demonstrate operations on the local router listening on localhost and the default AMQP port (5672). If you want to perform an operation on a different router in the router network, you must specify the necessary connection options. For more information, see [Connection Options](#) in the **qdmanage** man page.

To...

Use this command...

To...	Use this command...
View the router's listeners	<pre>qmanage query --type=listener</pre>
View the roles and ports on which the router is listening	<pre>qmanage query role port --type=listener</pre>
View the attributes configured for a listener	<pre>qmanage read --name=LISTENER_NAME</pre>
Create a listener	<pre>qmanage create --type=listener - -ATTRIBUTE=VALUE ...</pre>
Create multiple listeners	<p>1. Enter this command:</p> <pre>qmanage create --stdin</pre> <p>2. Configure the listeners using a JSON map:</p> <pre>[{"type"="listener", "ATTRIBUTE":"VALUE"...}, {"type"="listener", "ATTRIBUTE":"VALUE"...}...]</pre> <p>These commands use a JSON map to create two listeners.</p>
Update a listener	<pre>qmanage update --type=listener - -ATTRIBUTE=VALUE ...</pre>
Update multiple listeners	<p>1. Enter this command:</p> <pre>qmanage update --stdin</pre> <p>2. Configure the listeners using a JSON map:</p> <pre>[{"type"="listener", "ATTRIBUTE":"VALUE"...}, {"type"="listener", "ATTRIBUTE":"VALUE"...}...]</pre> <p>These commands use a JSON map to update two listeners.</p>
Delete an attribute from a listener	<pre>qmanage update --type=listener --ATTRIBUTE</pre>

To...	Use this command...
Delete a listener	<code>qdmange delete --name=LISTENER_NAME</code>

### 10.3.2.2. Managing Connectors

Connectors define how the router can connect to other endpoints in your messaging network, such as brokers and other routers. The following table lists the **qdmange** commands you can use to perform common operations on connectors.

For more information about the attributes you can use with these commands, see [connector](#) in the **qdrouterd.conf** man page.



#### NOTE

The commands in this table demonstrate operations on the local router listening on localhost and the default AMQP port (5672). If you want to perform an operation on a different router in the router network, you must specify the necessary connection options. For more information, see [Connection Options](#) in the qdmange man page.

To...	Use this command...
View the router's connectors	<code>qdmange query --type=connector</code>
View the roles and ports on which the router can connect to other endpoints	<code>qdmange query role port --type=connector</code>
If the router is connected to a broker, view the alternate URLs on which the router can connect to the broker if the primary connection fails	<code>qdmange query failoverUrls --type=connector --name=CONNECTOR_NAME</code>
View the attributes configured for a connector	<code>qdmange read --name=CONNECTOR_NAME</code>
Create a connector	<code>qdmange create --type=connector - -ATTRIBUTE=VALUE ...</code>



To...	Use this command...
Create multiple connectors	<ol style="list-style-type: none"> <li>1. Enter this command: <pre>qmanage create --stdin</pre> </li> <li>2. Configure the connectors using a JSON map: <pre>[{"type":"connector",   "ATTRIBUTE":"VALUE"...},  {"type":"connector",   "ATTRIBUTE":"VALUE"...}...]</pre> </li> </ol> <p>These commands use a JSON map to create two connectors.</p>
Update a connector	<pre>qmanage update --type=connector - -ATTRIBUTE=VALUE ...</pre>
Update multiple connectors	<ol style="list-style-type: none"> <li>1. Enter this command: <pre>qmanage update --stdin</pre> </li> <li>2. Configure the connectors using a JSON map: <pre>[{"type":"connector",   "ATTRIBUTE":"VALUE"...},  {"type":"connector",   "ATTRIBUTE":"VALUE"...}...]</pre> </li> </ol> <p>These commands use a JSON map to update two connectors.</p>
Delete an attribute from a connector	<pre>qmanage update --type=connector --ATTRIBUTE</pre>
Delete a connector	<pre>qmanage delete --name=CONNECTOR_NAME</pre>

### 10.3.3. Managing Security

AMQ Interconnect supports both SSL/TLS and SASL security protocols for encrypting and authenticating incoming and outgoing connections for your routers. You can use **qmanage** to view, create, update, and delete security policies for any router in your router network.

#### 10.3.3.1. Managing SSL/TLS Encryption and Authentication

AMQ Interconnect supports SSL/TLS for certificate-level encryption and mutual authentication. The following table lists the common **qmanage** commands you can use to secure incoming and outgoing connections for a router in your router network.

For more information about the attributes you can use with these commands, see [sslProfile](#) and [listener](#) in the **qdrouterd.conf** man page.



## NOTE

The commands in this table demonstrate operations on the local router listening on localhost and the default AMQP port (5672). If you want to perform an operation on a different router in the router network, you must specify the necessary connection options. For more information, see [Connection Options](#) in the qdmanage man page.

To...	Use this command...
View the router's SSL/TLS configuration	<code>qdmanage query --type=sslProfile</code>
Set up SSL/TLS for the router	<code>qdmanage create --type=sslProfile --name=NAME - -ATTRIBUTE=VALUE ...</code>
Add SSL/TLS encryption to an incoming connection	<code>qdmanage update --name=LISTENER_NAME -- sslProfile=NAME --requireSsl=yes</code>
Change SSL/TLS encryption on an incoming connection	<code>qdmanage update --name=LISTENER_NAME - -ATTRIBUTE=VALUE ...</code>
Add SSL/TLS client authentication to an incoming connection	<code>qdmanage update --name=LISTENER_NAME -- authenticatePeer=yes</code>
Remove SSL/TLS client authentication from an incoming connection	<code>qdmanage update --name=LISTENER_NAME -- authenticatePeer=no</code>
Add SSL/TLS client authentication to an outgoing connection	<code>qdmanage update --name=CONNECTOR_NAME -- sslProfile=NAME</code>
Remove SSL/TLS client authentication from an outgoing connection	<code>qdmanage update --name=CONNECTOR_NAME -- sslProfile</code>
Delete an SSL profile	<code>qdmanage delete --name=SSL_PROFILE_NAME</code>

### 10.3.3.2. Managing SASL Encryption and Authentication

AMQ Interconnect supports SASL for authentication and payload encryption. The following table lists the common **qmanage** commands you can use to secure incoming and outgoing connections for a router in your router network.

For more information about the attributes you can use with these commands, see [router](#) and [listener](#) in the **qdrouterd.conf** man page.



#### NOTE

The commands in this table demonstrate operations on the local router listening on localhost and the default AMQP port (5672). If you want to perform an operation on a different router in the router network, you must specify the necessary connection options. For more information, see [Connection Options](#) in the qmanage man page.

To...	Use this command...
Set up SASL for the router	<pre>qmanage update --type=router -- saslConfigDir=PATH --saslConfigName=NAME</pre>
Add SASL authentication to an incoming connection	<pre>qmanage update --name=LISTENER_NAME -- authenticatePeer=yes --saslMechanisms=MECHANISMS</pre>
Change SASL mechanisms for an incoming connection	<pre>qmanage update --name=LISTENER_NAME -- saslMechanisms=MECHANISMS</pre>
Add SASL authentication to an outgoing connection	<pre>qmanage update --name=CONNECTOR_NAME -- saslMechanisms=MECHANISMS -- saslUsername=USERNAME --saslPassword=PASSWORD</pre>
Change SASL mechanisms for an outgoing connection	<pre>qmanage update --name=CONNECTOR_NAME -- saslMechanisms=MECHANISMS</pre>
Add SASL payload encryption to an incoming connection	<pre>qmanage update --name=LISTENER_NAME -- requireEncryption=yes -- saslMechanisms=MECHANISMS</pre>
Change SASL mechanisms for an incoming connection	<pre>qmanage update --name=LISTENER_NAME -- saslMechanisms=MECHANISMS</pre>

To...	Use this command...
Remove SASL payload encryption from an incoming connection	<pre>qdmmanage update --name=LISTENER_NAME --requireEncryption=no --saslMechanisms</pre>
Delete a SASL configuration	<pre>qdmmanage update --type=router --saslConfigDir --saslConfigName</pre>

### 10.3.4. Managing Routing

AMQ Interconnect supports both message routing and link routing for distributing messages between senders and receivers. You can use **qdmmanage** to view how addresses and link routes are configured in your environment, and define how a router should distribute messages.

#### 10.3.4.1. Managing Message Routing

Message routing involves configuring addresses to define how AMQ Interconnect should distribute messages. The following table lists the common **qdmmanage** commands you can use to configure addresses for a router in your router network.

For more information about the attributes you can use with these commands, see [address](#) and [autolink](#) in the **qdrouterd.conf** man page.



#### NOTE

The commands in this table demonstrate operations on the local router listening on localhost and the default AMQP port (5672). If you want to perform an operation on a different router in the router network, you must specify the necessary connection options. For more information, see [Connection Options](#) in the **qdmmanage** man page.

To...	Use this command...
View addresses	<pre>qdmmanage query --type=address</pre> <pre>qdmmanage read --name=ADDRESS_NAME</pre>
View address distribution patterns	<pre>qdmmanage query prefix distribution --type=address</pre>
View waypoints to broker queues	<pre>qdmmanage query prefix --type=address --waypoint=yes</pre>

To...	Use this command...
View autolinks	<pre>qmanage query --type=autolink</pre>
Set a distribution pattern for an address	<pre>qmanage create --type=address -- prefix=ADDRESS_PREFIX -- distribution=DISTRIBUTION_PATTERN ...</pre>
Set distribution patterns for multiple addresses	<ol style="list-style-type: none"> <li>1. Enter this command: <pre>qmanage create --stdin</pre> </li> <li>2. Configure the addresses using a JSON map: <pre>[{"type":"address", "prefix":"ADDRESS_PREFIX", "distribution":"DISTRIBUTION_PATTERN", "ATTRIBUTE":"VALUE", ...}, {"type":"address", "prefix":"ADDRESS_PREFIX", "distribution":"DISTRIBUTION_PATTERN", "ATTRIBUTE":"VALUE", ...} ...]</pre> </li> </ol> <p>These commands configure two addresses.</p>
Connect an address to a broker queue	<ol style="list-style-type: none"> <li>1. Enter this command: <pre>qmanage create --stdin</pre> </li> <li>2. Create an address waypoint, an incoming autolink, and an outgoing autolink: <pre>[{"type":"address", "prefix":"ADDRESS_PREFIX", "waypoint":"yes"}, {"type":"autolink", "addr":"ADDRESS_NAME", "connection":"CONNECTOR/LISTENER_NAME", "direction":"in"}, {"type":"autolink", "addr":"ADDRESS_NAME", "connection":"CONNECTOR/LISTENER_NAME", "direction":"out"}]</pre> </li> </ol>
Update an address configuration	<pre>qmanage update --name=ADDRESS_NAME - -ATTRIBUTE=VALUE ...</pre>

To...	Use this command...
Update an autolink	<pre>qmanage update --name=AUTOLINK_NAME - -ATTRIBUTE=VALUE ...</pre>
Delete an address configuration	<pre>qmanage delete --name=ADDRESS_NAME</pre>
Delete an autolink	<pre>qmanage delete --name=AUTOLINK_NAME</pre>

### 10.3.4.2. Managing Link Routing

A link route is a chain of links between a sender and receiver that provides a private messaging path. The following table lists the common **qmanage** commands you can use to view, create, update, and delete link routes.

For more information about the attributes you can use with these commands, see the [linkRoute](#) in the **qdrouterd.conf** man page.



#### NOTE

The commands in this table demonstrate operations on the local router listening on localhost and the default AMQP port (5672). If you want to perform an operation on a different router in the router network, you must specify the necessary connection options. For more information, see [Connection Options](#) in the qmanage man page.

To...	Use this command...
View link routes	<pre>qmanage query --type=linkRoute</pre> <pre>qmanage read --name=LINK_ROUTE_NAME</pre>

To...	Use this command...
Create a link route	<ol style="list-style-type: none"> <li>1. Enter this command: <pre>qmanage create --stdin</pre> </li> <li>2. Create an incoming and outgoing link route: <pre>[{"type":"linkRoute",   "prefix":"ADDRESS_PREFIX",   "connection":"CONNECTOR/LISTENER_NAME",   "direction":"in", ...},  {"type":"linkRoute",   "prefix":"ADDRESS_PREFIX",   "connection":"CONNECTOR/LISTENER_NAME",   "direction":"out", ...}]</pre> </li> </ol>
Update a link route	<pre>qmanage update --name=LINK_ROUTE_NAME - -ATTRIBUTE=VALUE ...</pre>
Delete a link route	<pre>qmanage delete --name=INCOMING_LINK_ROUTE_NAME qmanage delete --name=OUTGOING_LINK_ROUTE_NAME</pre>

### 10.3.5. Managing Logging

AMQ Interconnect logs are broken into different categories called logging modules. Each module provides important information about a particular aspect of a router. The following table lists the common **qmanage** commands you can use to view and change the configuration of a logging module.

For more information about the attributes you can use with these commands, see [log](#) in the **qdrouterd.conf** man page.



#### NOTE

The commands in this table demonstrate operations on the local router listening on localhost and the default AMQP port (5672). If you want to perform an operation on a different router in the router network, you must specify the necessary connection options. For more information, see [Connection Options](#) in the **qmanage** man page.

To...	Use this command...
View the logging configuration	<pre>qmanage query --type=log</pre>

To...	Use this command...
View the logging configuration for a logging module	<pre>qmanage read --type=log -- module=LOGGING_MODULE_NAME</pre>
Set the default logging configuration	<pre>qmanage create --type=log --module=DEFAULT -- enable=LOGGING_LEVEL --includeTimestamp=yes - -ATTRIBUTE=VALUE</pre>
Enable logging for a logging module	<pre>qmanage create --type=log -- module=LOGGING_MODULE_NAME -- enable=LOGGING_LEVEL --ATTRIBUTE=VALUE ...</pre>
Change the logging configuration for a logging module	<pre>qmanage update --type=log -- module=LOGGING_MODULE_NAME --ATTRIBUTE=VALUE ...</pre>



## CHAPTER 11. RELIABILITY

In general, in a broker based architecture, the reliability feature is strictly related to the "store and forward" mechanism offered by each broker. Thanks to persistent journals, a broker can offer fault tolerance thus avoiding message loss; of course, it is not so true when messages are stored only in a volatile memory.

This is completely different using AMQ Interconnect, because each router neither takes ownership of messages nor stores them in a persistent storage. In this case, the reliability feature is offered by **path redundancy** which provides the possibility to reach the destination on different paths through the router network. In normal conditions, the best path is always chosen in terms of lowest cost but, when one or more routers go down, the topology is revisited by all remained routers and new paths are processed in order to reach always each destination. Of course, it means that the reliability is strictly related to the network topology the user chooses for his solution.

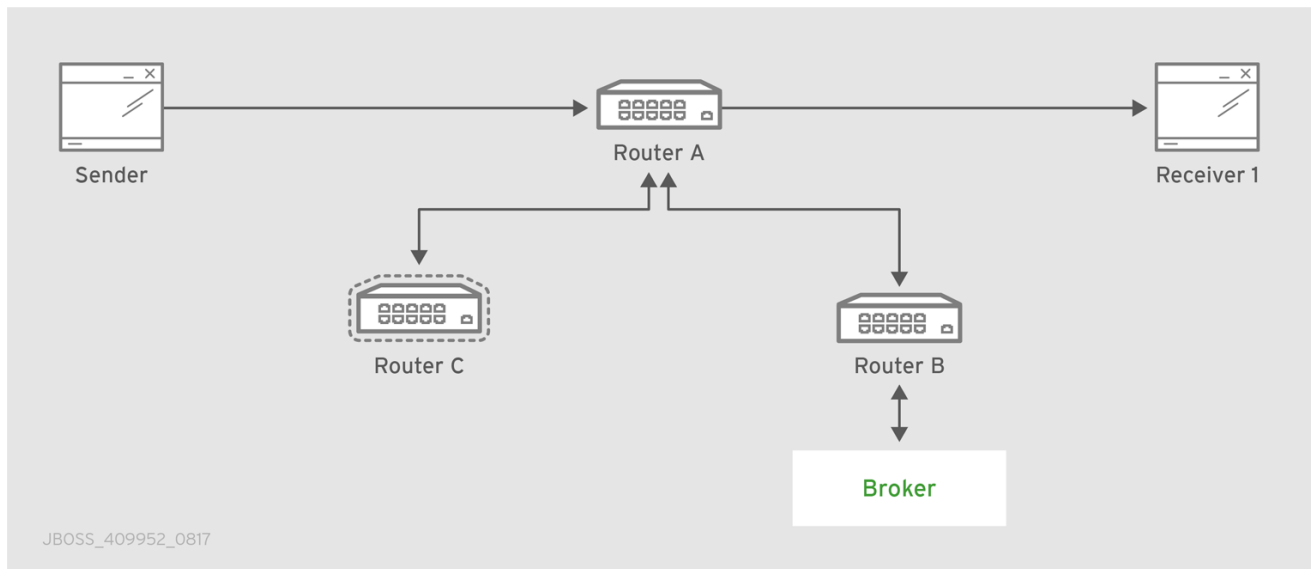
Because a solution based on AMQ Interconnect could be made not only by routers but by brokers too, the reliability is improved with persistent storage on them which add not only fault tolerance but temporal decoupling as well; without "store and forward" feature offered by brokers, the temporal decoupling is not possible only with routers and direct peers, both senders and receivers; the receiver must be online at same time of the sender in order to receive messages.

### 11.1. PATH REDUNDANCY

Offering path redundancy means designing the network topology in a way that even when one or more routers go down or even connections between them, each destination is always reachable following alternate paths through the routers that are still part of the network.

Consider the following simple scenario :

- a network with three routers "Router.A", "Router.B" and "Router.C".
- the "Router.A" is connected to both "Router.B" and "Router.C".
- the "Router.C" is connected to the "Router.B".
- all three routers listen for client connections.
- a sender client connects to the "Router.A" in order to send messages to a receiver client.
- a receiver client connects to the "Router.B" initially in order to receive messages from the sender peer.

**Figure 11.1. Path Redundancy Enabled Topology**

The "Router.A" configuration is something like following.

```

router {
    mode: interior
    id: Router.A
}

listener {
    host: 0.0.0.0
    port: 6000
    authenticatePeer: no
}

connector {
    name: INTER_ROUTER_B
    addr: 127.0.0.1
    port: 5001
    role: inter-router
}

connector {
    name: INTER_ROUTER_C
    addr: 127.0.0.1
    port: 5002
    role: inter-router
}

```

There is only one *listener* in order to accept client connections and two *connector* entities for connecting to the other two routers.

The "Router.B" configuration is the following.

```

router {
    mode: interior
    id: Router.B
}

```

```

listener {
    addr: 0.0.0.0
    port: 5001
    authenticatePeer: no
    role: inter-router
}

listener {
    host: 0.0.0.0
    port: 6001
    authenticatePeer: no
}

```

It has two *listener* entities in order to listen for connections from clients and from other routers in the network (in this case from the "Router.A" and "Router.C").

Finally, quite similar is the "Router.C" configuration.

```

router {
    mode: interior
    id: Router.C
}

listener {
    addr: 0.0.0.0
    port: 5002
    authenticatePeer: no
    role: inter-router
}

listener {
    host: 0.0.0.0
    port: 6002
    authenticatePeer: no
}

connector {
    name: INTER_ROUTER_B
    addr: 127.0.0.1
    port: 5001
    role: inter-router
}

```

It has two *listener* entities in order to listen for connections from clients and from other routers in the network (in this case from the "Router.A") and finally it has a *connector* (for connecting to the "Router.B")

Consider a sender client connected to "Router.A" and attached to **my\_address** address which start to send messages (that is, 10 messages) and a receiver client connected to the "Router.B" and attached to the same address.

Starting the receiver, it waits for messages with no output on the console.

```
$ sudo python simple_recv.py -a localhost:6001/my_queue -m 10
```

Starting the sender, all the messages flow through "Router.A" and "Router.B" reaching the receiver; at this point the messages are all confirmed at sender side.

```
$ sudo python simple_send.py -a localhost:6001/my_queue -m 10
all messages confirmed
```

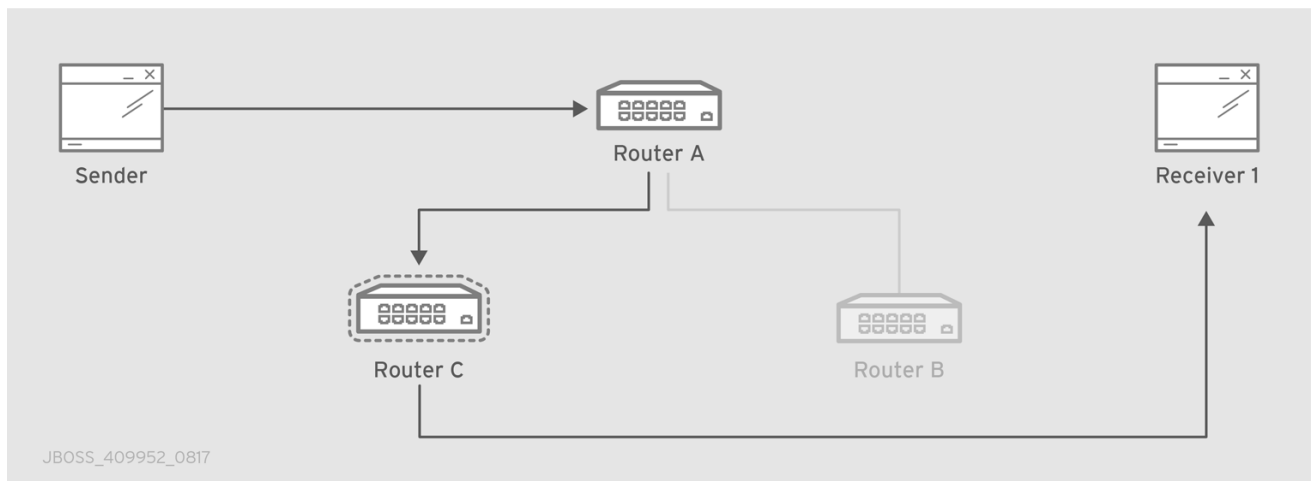
At same time, the receivers shows the messages received through the "Router.B".

```
{u'sequence': 1L}
{u'sequence': 2L}
{u'sequence': 3L}
{u'sequence': 4L}
{u'sequence': 5L}
{u'sequence': 6L}
{u'sequence': 7L}
{u'sequence': 8L}
{u'sequence': 9L}
{u'sequence': 10L}
```

The path redundancy is provided by the other available path through the "Router.A", "Router.C" and then "Router.B". It means that if the connection between "Router.A" and "Router.B" goes down, the alternative path is used to reach the receiver.

Now, consider a fault on the "Router.B"; the receiver is not reachable anymore on that path but it can connect to the "Router.C" in order to continue to receive messages from the sender which does not know what's happened and it can continue to send messages to the "Router.A" in order to reach the receiver.

**Figure 11.2. Path Redundancy after Router Failure**



The receiver is still reachable in order to get messages from the sender as displayed in the console output.

```
$ sudo python simple_recv.py -a localhost:6002/my_queue -m 10
{u'sequence': 1L}
{u'sequence': 2L}
{u'sequence': 3L}
{u'sequence': 4L}
{u'sequence': 5L}
{u'sequence': 6L}
{u'sequence': 7L}
```

```
{u'sequence': 8L}
{u'sequence': 9L}
{u'sequence': 10L}
```

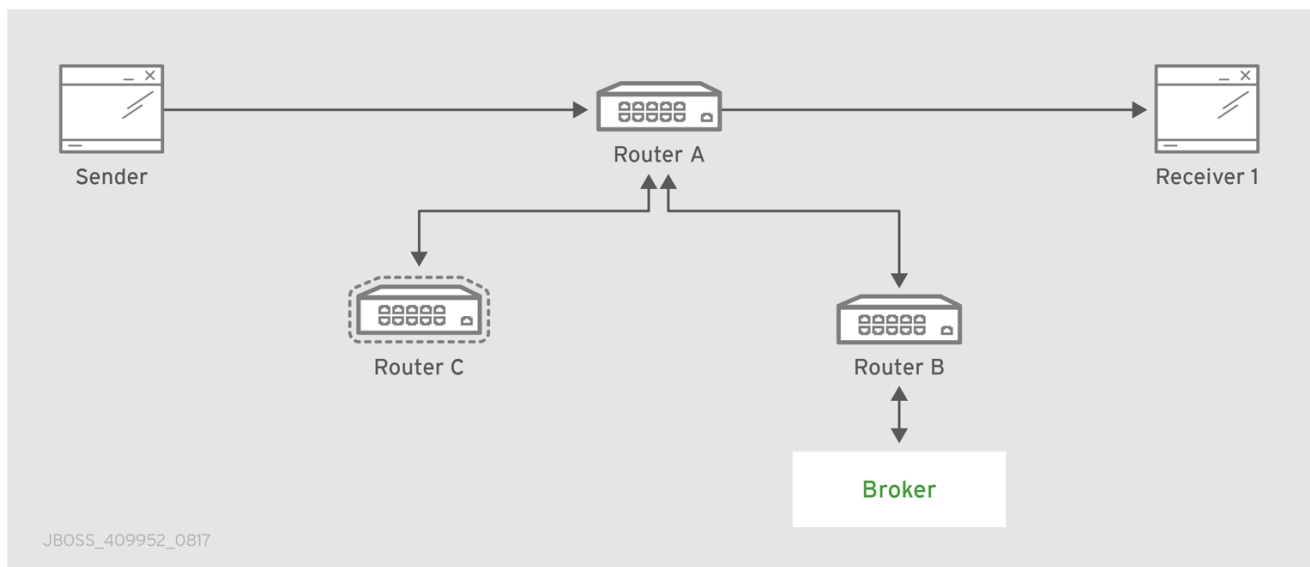
## 11.2. PATH REDUNDANCY AND TEMPORAL DECOUPLING

In order to have temporal decoupling in a solution based on AMQ Interconnect, adding one or more brokers is a must for its "store and forward" feature. Choosing the right topology, it is possible to have a solution which offers reliability with both path redundancy and permanent storing for messages.

Consider the following simple scenario :

- a network with three routers "Router.A", "Router.B" and "Router.C" and finally a broker.
- the "Router.A" is connected to both "Router.B" and "Router.C".
- initially only the "Router.B" is connected to the broker.
- all three routers listen for client connections.
- a sender client connects to the "Router.A" in order to send messages to a queue in the broker.
- a receiver client connects to the "Router.A" in order to get messages from the queue in the broker.

**Figure 11.3. Path Redundancy and Temporal Decoupling Enabled Topology**



The receiver client can be offline when the sender starts to send messages because they'll be stored into the queue permanently; coming back online, the receiver can get messages from the queue itself without message loss.

The "Router.A" configuration is something like following.

```
router {
    mode: interior
    id: Router.A
}

listener {
```

```

    host: 0.0.0.0
    port: 6000
    authenticatePeer: no
}

connector {
    name: INTER_ROUTER_B
    addr: 127.0.0.1
    port: 5001
    role: inter-router
}

connector {
    name: INTER_ROUTER_C
    addr: 127.0.0.1
    port: 5002
    role: inter-router
}

address {
    prefix: my_queue
    waypoint: yes
}

```

It has a *listener* for accepting incoming connections from clients and two *connector* entities in order to connect to the other routers. The queue named **my\_queue** on the broker is exposed by a waypoint.

The "Router.B" configuration is the following.

```

router {
    mode: interior
    id: Router.B
}

listener {
    addr: 0.0.0.0
    port: 5001
    authenticatePeer: no
    role: inter-router
}

listener {
    host: 0.0.0.0
    port: 6001
    authenticatePeer: no
}

connector {
    name: BROKER
    addr: 127.0.0.1
    port: 5672
    role: route-container
}

address {
    prefix: my_queue
}

```

```

        waypoint: yes
    }

    autoLink {
        addr: my_queue
        connection: BROKER
        direction: in
    }

    autoLink {
        addr: my_queue
        connection: BROKER
        direction: out
    }

```

It can accept incoming connections from clients and from other routers (in this case the "Router.A") and connects to the broker. The queue named **my\_queue** on the broker is exposed by a waypoint with the related auto-links in both directions in order to send and receive messages to/from the queue itself.

Finally, the simple "Router.C" configuration.

```

router {
    mode: interior
    id: Router.C
}

listener {
    addr: 0.0.0.0
    port: 5002
    authenticatePeer: no
    role: inter-router
}

listener {
    host: 0.0.0.0
    port: 6002
    authenticatePeer: no
}

```

It can accept incoming connections from clients and from other routers (in this case the "Router.A"). Initially there is no connection between this router and the broker.

First of all, thanks to the broker and its "store and forward" feature, the sender can connect to the "Router.A" and start to send messages even if the receiver is not online in that moment. Using the Python sample from the Qpid Proton library, the console output is like following.

```

$ sudo python simple_send.py -a localhost:6000/my_queue -m 10
all messages confirmed

```

All messages are confirmed because they reached the queue inside the broker through "Router.A" and "Router.B"; it is confirmed using the **qdstat** tool.

```

$ sudo qdstat -b localhost:6001 -a
Router Addresses
class  addr                                phs  distrib  in-proc  local  remote

```

```
cntnr  in  out  thru  to-proc  from-proc
```

```
=====
=====
  local  $_management_internal      closest    1      0      0
0      0    0    0    0      0
  local  $displayname                closest    1      0      0
0      0    0    0    0      0
  mobile $management                0 closest    1      0      0
0      1    0    0    1      0
  local  $management                closest    1      0      0
0      0    0    0    0      0
  router Router.A                    closest    0      0      1
0      0    0    6    0      6
  router Router.C                    closest    0      0      1
0      0    0    4    0      4
  mobile my_queue                    1 balanced    0      0      0
0      0    0    0    0      0
  mobile my_queue                    0 balanced    0      1      0
0      0   10    0    0      0
  local  qdhello                      flood      1      1      0
0      0    0    0   97     117
  local  qdrouter                     flood      1      0      0
0      0    0    0    7      0
  topo   qdrouter                     flood      1      0      2
0      0    0    8   13      9
  local  qdrouter.ma                  multicast  1      0      0
0      0    0    0    2      0
  topo   qdrouter.ma                  multicast  1      0      2
0      0    0    0    0      1
  local  temp.7f2u0zv9_U6QC5e         closest    0      1      0
0      0    0    0    0      0
```

For the "Router.B", there are 10 messages as output (from the router to the broker) on the **my\_queue** address.

Starting the receiver connected to the "Router.A", it gets all the available messages from the queue.

```
$ sudo python simple_recv.py -a localhost:6000/my_queue -m 10
{u'sequence': 1L}
{u'sequence': 2L}
{u'sequence': 3L}
{u'sequence': 4L}
{u'sequence': 5L}
{u'sequence': 6L}
{u'sequence': 7L}
{u'sequence': 8L}
{u'sequence': 9L}
{u'sequence': 10L}
```

Using the **qdstat** tool on the "Router.B" another time, the output is like following.

```
$ sudo qdstat -b localhost:6001 -a
Router Addresses
class  addr                                phs  distrib  in-proc  local  remote
```



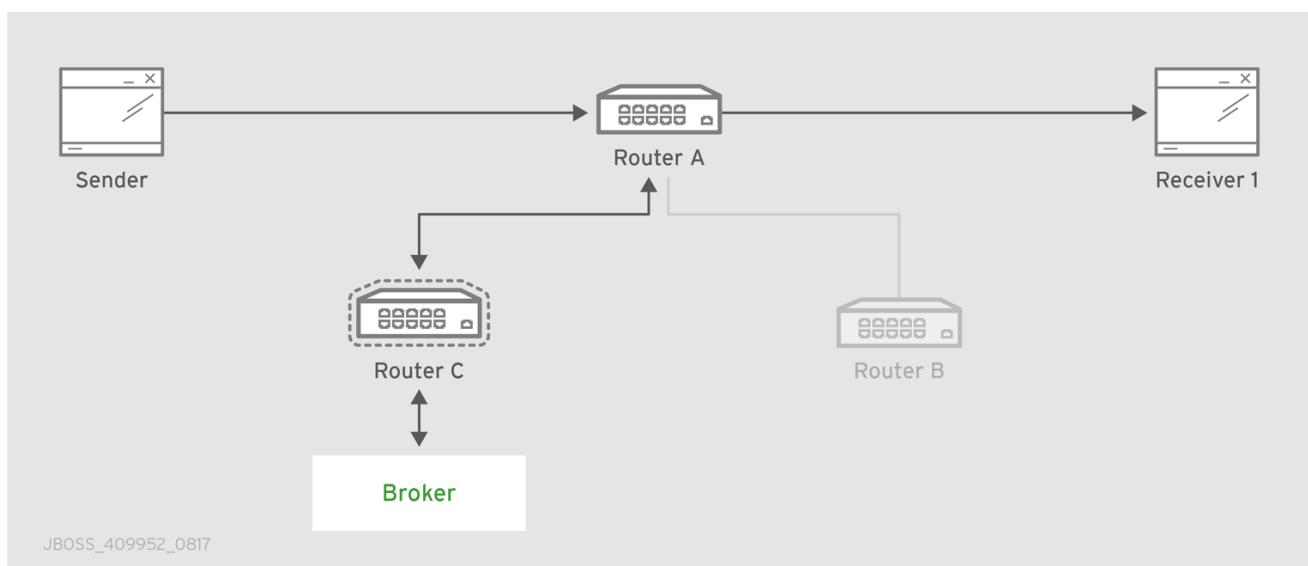
```
cntnr in out thru to-proc from-proc
```

```
=====
=====
local  $_management_internal      closest  1      0      0
0      0      0      0      0      0
local  $displayname                closest  1      0      0
0      0      0      0      0      0
mobile $management                0      closest  1      0      0
0      2      0      0      2      0
local  $management                closest  1      0      0
0      0      0      0      0      0
router Router.A                   closest  0      0      1
0      0      0      6      0      6
router Router.C                   closest  0      0      1
0      0      0      4      0      4
mobile my_queue                   1      balanced 0      0      0
0      10     0      10     0      0
mobile my_queue                   0      balanced 0      1      0
0      0      10     0      0      0
local  qdhello                    flood    1      1      0
0      0      0      0      156    182
local  qdrouter                   flood    1      0      0
0      0      0      0      7      0
topo   qdrouter                   flood    1      0      2
0      0      0      10     18     11
local  qdrouter.ma                multicast 1      0      0
0      0      0      0      2      0
topo   qdrouter.ma                multicast 1      0      2
0      0      0      0      2      1
local  temp.Xov_ZUcyti3jjXY       closest  0      1      0
0      0      0      0      0      0
```

For the "Router.B", there are 10 messages as input (from the broker to the router) on the **my\_queue** address.

Now, consider a fault on the "Router.B"; in this case the broker is not reachable but it is possible to set up path redundancy through the "Router.C".

**Figure 11.4. Path Redundancy and Temporal Decoupling after Router Failure**



Using the **qmanage** tool, it is possible to configure the waypoint on **my\_queue** address, the related auto-links in both directions and finally the *connector* instance in order to enable the connection to the broker.

```
$ sudo qmanage -b localhost:6002 create --stdin
[
{ "type": "connector", "name": "BROKER", "port": 5672, "role": "route-
container" },
{ "type": "address", "prefix": "my_queue", "waypoint": "yes" },
{ "type": "autoLink", "addr": "my_queue", "connection": "BROKER",
"direction": "in" },
{ "type": "autoLink", "addr": "my_queue", "connection": "BROKER",
"direction": "out" }
]
[
{
  "verifyHostname": true,
  "stripAnnotations": "both",
  "name": "BROKER",
  "allowRedirect": true,
  "idleTimeoutSeconds": 16,
  "maxFrameSize": 65536,
  "host": "127.0.0.1",
  "cost": 1,
  "role": "route-container",
  "maxSessions": 32768,
  "type": "org.apache.qpid.dispatch.connector",
  "port": "5672",
  "identity": "connector/127.0.0.1:5672:BROKER",
  "addr": "127.0.0.1"
},
{
  "name": null,
  "prefix": "my_queue",
  "ingressPhase": 0,
  "waypoint": false,
  "distribution": "balanced",
  "type": "org.apache.qpid.dispatch.router.config.address",
  "identity": "7",
  "egressPhase": 0
},
{
  "addr": "my_queue",
  "name": null,
  "linkRef": null,
  "type": "org.apache.qpid.dispatch.router.config.autoLink",
  "operStatus": "inactive",
  "connection": "BROKER",
  "direction": "in",
  "phase": 1,
  "lastError": null,
  "externalAddr": null,
  "identity": "8",
  "containerId": null
},
{
```

```

    "addr": "my_queue",
    "name": null,
    "linkRef": null,
    "type": "org.apache.qpid.dispatch.router.config.autoLink",
    "operStatus": "inactive",
    "connection": "BROKER",
    "direction": "out",
    "phase": 0,
    "lastError": null,
    "externalAddr": null,
    "identity": "9",
    "containerId": null
  }
]

```

The "Router.C" configuration changes in the same way as "Router.B". It can accept incoming connections from clients and from other routers (in this case the "Router.A") and connects to the broker. The queue named **my\_queue** on the broker is exposed by a waypoint with the related auto-links in both directions in order to send and receive messages to/from the queue itself.

At this point, the sender can connect to the "Router.A" for sending messages to the queue in the broker thanks to the "Router.C".

```

$ sudo python simple_send.py -a localhost:6000/my_queue -m 10
all messages confirmed

```

All messages are confirmed because they reached the queue inside the broker through "Router.A" and "Router.C"; it is confirmed using the **qdstat** tool.

```

$ sudo qdstat -b localhost:6002 -a
Router Addresses
  class  addr                phs  distrib  in-proc  local  remote
cntnr  in  out  thru  to-proc  from-proc
=====
=====
  local  $_management_internal    closest    1      0      0
0      0    0    0    1      1
  local  $displayname             closest    1      0      0
0      0    0    0    0      0
  mobile $management          0    closest    1      0      0
0      5    0    0    5      0
  local  $management             closest    1      0      0
0      0    0    0    0      0
  router Router.A           closest    0      0      1
0      0    0    5    0      5
  mobile my_queue           0    balanced    0      1      0
0      0   10    0    0      0
  mobile my_queue           1    balanced    0      0      0
0      0    0    0    0      0
  local  qdhello               flood      1      1      0
0      0    0    0   665    647
  local  qdrouter              flood      1      0      0
0      0    0    0    8      0
  topo   qdrouter              flood      1      0      1

```

```

0      0      0      31      52      32
  local  qdrouter.ma      multicast  1      0      0
0      0      0      0      1      0
  topo  qdrouter.ma      multicast  1      0      1
0      0      0      1      2      1
  local  temp.k6UMaS4P0JmtSlL      closest  0      1      0
0      0      0      0      0      0

```

For the "Router.C", there are 10 messages as output (from the router to the broker) on the **my\_queue** address.

Starting the receiver connected to the "Router.A", it gets all the available messages from the queue.

```

$ sudo python simple_recv.py -a localhost:6000/my_queue -m 10
{u'sequence': 1L}
{u'sequence': 2L}
{u'sequence': 3L}
{u'sequence': 4L}
{u'sequence': 5L}
{u'sequence': 6L}
{u'sequence': 7L}
{u'sequence': 8L}
{u'sequence': 9L}
{u'sequence': 10L}

```

Using the **qdstat** tool on the "Router.C" another time, the output is like following.

```

$ sudo qdstat -b localhost:6002 -a
Router Addresses
  class  addr                phs  distrib  in-proc  local  remote
cntnr  in  out  thru  to-proc  from-proc

=====
  local  $_management_internal      closest  1      0      0
0      0      0      0      1      1
  local  $displayname                  closest  1      0      0
0      0      0      0      0      0
  mobile $management                0      closest  1      0      0
0      6      0      0      6      0
  local  $management                  closest  1      0      0
0      0      0      0      0      0
  router Router.A                  closest  0      0      1
0      0      0      5      0      5
  mobile my_queue                  0      balanced  0      1      0
0      0      10     0      0      0
  mobile my_queue                  1      balanced  0      0      0
0      10     0      10     0      0
  local  qdhello                      flood    1      1      0
0      0      0      0      746    726
  local  qdrouter                      flood    1      0      0
0      0      0      0      8      0
  topo  qdrouter                      flood    1      0      1
0      0      0      34     55     35
  local  qdrouter.ma                  multicast 1      0      0

```

[illegible]

For the "Router.C", there are 10 messages as input (from the broker to the router) on the **my\_queue** address.

### 11.3. SHARDED QUEUE

Every broker has limits in terms of queue size but in order to overcome this problem, one possible solution is "sharding" queues : in that way a single queue is divided in more "shards" (chunks) each on a different broker. It means that such solution needs more than one broker instance in order to host a shard on each of them. Of course, a sender connected to one of these brokers can send messages to the shard hosted only on that broker. At same time, a receiver connected to a broker can get messages from the shard that is hosted on that broker and can not see available messages in the shards hosted on the other brokers, even if they are all parts of the same queue.



## NOTE

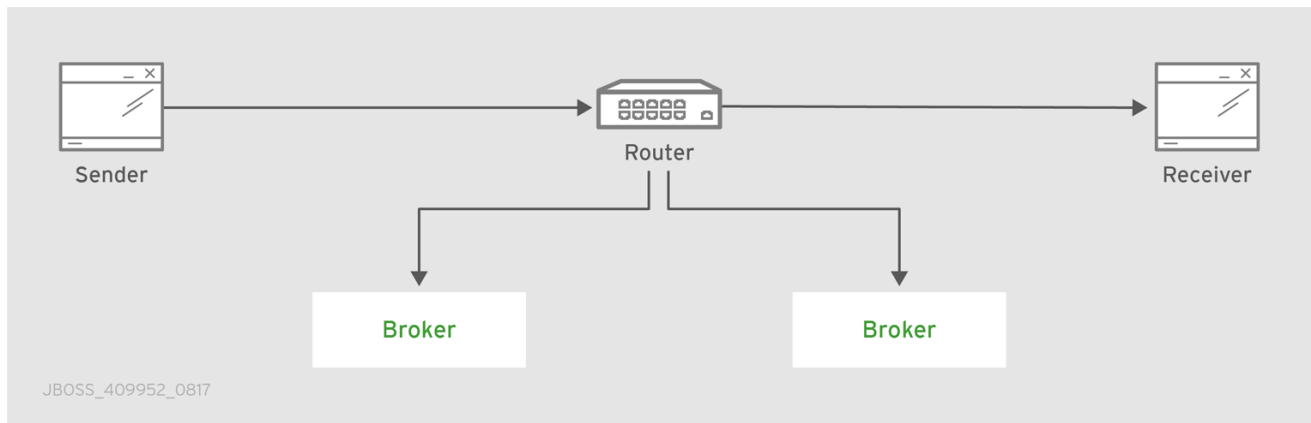
Even if speaking about shards it is obvious that they are real queues all with same name but on different brokers. The "shard" concept is an abstract one because finally a shard is a real queue stored on a broker.

The big problem in this scenario, designed only with brokers, is that a receiver can be stuck on an empty shard without reading any messages while the shards on the other brokers have messages to deliver. It is a real problem because the receiver is interested in receiving messages from the whole queue and it does not take care if it is shared or not. Because of this problem, the receiver sees the queue as empty even if it is not so true due to the sharding and the messages available on the other shards.

The above problem can be solved adding a AMQ Interconnect instance in the network in front of the brokers and leverage on its waypoint feature with related auto-links.

Consider the following simple scenario :

- a network with one router "Router.A" and two brokers.
- the "Router.A" listens for clients connections and it is connected to both brokers.
- the brokers host shards for a queue; each broker has one shard.
- a sender client connects to the "Router.A" in order to send messages to the queue.
- a receiver client connects to the "Router.A" in order to get messages from the queue.

**Figure 11.5. Sharded Queue Enabled Topology**

With such solution and connecting to the "Router.A", sender and receiver do not know anything about sharding; they want send and receive messages to/from the whole queue that is the only thing they are aware of. They are both connected to the router and see only one address (related to the queue).

The "Router.A" configuration is something like following.

```

router {
    mode: standalone
    id: Router.A
}

listener {
    host: 0.0.0.0
    port: 6000
    authenticatePeer: no
}

connector {
    name: BROKER1
    addr: 127.0.0.1
    port: 5672
    role: route-container
}

connector {
    name: BROKER2
    addr: 127.0.0.1
    port: 5673
    role: route-container
}

address {
    prefix: my_queue
    waypoint: yes
}

autoLink {
    addr: my_queue
    connection: BROKER1
    direction: in
}

```

```

autoLink {
    addr: my_queue
    connection: BROKER1
    direction: out
}

autoLink {
    addr: my_queue
    connection: BROKER2
    direction: in
}

autoLink {
    addr: my_queue
    connection: BROKER2
    direction: out
}

```

The router has a *listener* for incoming connection from clients and two *connector* instances in order to connect to both brokers. The whole queue is named **my\_queue** hosted in terms of shards on both brokers and the router is configured with a waypoint for that address. Finally, there are two auto-links in both directions for that queue on both brokers.

Using the Python sample from the Qpid Proton library, the sender can connect to the "Router.A" and start to send messages to the queue; the console output is like following.

```

$ sudo python simple_send.py -a localhost:6000/my_queue -m 10
all messages confirmed

```

All messages are confirmed because they reached the queue and, thanks to the default **balanced** distribution on the address, the messages are delivered to both shards on the brokers (5 messages per shard). Using the **qdstat** tool on the router, the distribution is clear.

```

$ sudo qdstat -b localhost:6000 -l
Router Links
  type      dir  conn id  id  peer  class  addr  phs
cap undel  unsettled deliveries admin oper
=====
=====
  endpoint  in   1      6      mobile  my_queue  1
250  0      0      0      enabled up
  endpoint  out  1      7      mobile  my_queue  0
250  0      0      5      enabled up
  endpoint  in   2      8      mobile  my_queue  1
250  0      0      0      enabled up
  endpoint  out  2      9      mobile  my_queue  0
250  0      0      5      enabled up
  endpoint  in   8      19     mobile  $management  0
250  0      0      1      enabled up
  endpoint  out  8      20     local   temp.qCGHruCa4UIvYrS
250  0      0      0      enabled up

```

There are the **out** links (from router to brokers) for the **my\_queue** address (*id* values **7** and **9**) which have each 5 deliveries. It shows messages distributed across brokers and related shards for the queue; it is confirmed by the different connections they are tied (*conn id* values **1** and **2**).

Starting the receiver connected to the "Router.A", it gets all the available messages from the queue.

```
$ sudo python simple_recv.py -a localhost:6000/my_queue -m 10
{u'sequence': 1L}
{u'sequence': 2L}
{u'sequence': 3L}
{u'sequence': 4L}
{u'sequence': 5L}
{u'sequence': 6L}
{u'sequence': 7L}
{u'sequence': 8L}
{u'sequence': 9L}
{u'sequence': 10L}
```

As for the sender, they are received through both the brokers and related shards. it is confirmed using the **qdstat** tool.

```
$ sudo qdstat -b localhost:6000 -l
Router Links
  type      dir  conn id  id peer  class  addr  pbs
cap undel unsettled deliveries admin oper
=====
=====
  endpoint  in   1    6    mobile my_queue  1
250  0      0      5    enabled up
  endpoint  out  1    7    mobile my_queue  0
250  0      0      5    enabled up
  endpoint  in   2    8    mobile my_queue  1
250  0      0      5    enabled up
  endpoint  out  2    9    mobile my_queue  0
250  0      0      5    enabled up
  endpoint  in  10   22   mobile $management  0
250  0      0      1    enabled up
  endpoint  out  10   23   local  temp.HT+f3ZilGP5o3wo
250  0      0      0    enabled up
```

There are the **in** links (from brokers to router) for the **my\_queue** address (*id* values **6** and **8**) which have each 5 deliveries. It shows messages distributed across brokers and related shards for the queue; it is confirmed by the different connections they are tied (*conn id* values **1** and **2**).

One disadvantage of sharded queues is that the receiver might receive messages "out of order" even with very good performance.



## APPENDIX A. USING CYRUS SASL TO PROVIDE AUTHENTICATION

AMQ Interconnect uses the Cyrus SASL library for SASL authentication. Therefore, if you want to use SASL, you must set up the Cyrus SASL database and configure it.

### A.1. GENERATING A SASL DATABASE

To generate a SASL database to store credentials, enter the following command:

```
$ sudo saslpasswd2 -c -f SASL_DATABASE_NAME.sasldb -u DOMAIN_NAME
USER_NAME
```

This command creates or updates the specified SASL database, and adds the specified user name to it. The command also prompts you for the user name's password.

The full user name is the user name you entered plus the domain name (**USER\_NAME@DOMAIN\_NAME**). Providing a domain name is not required when you add a user to the database, but if you do not provide one, a default domain will be added automatically (the hostname of the machine on which the tool is running). For example, in the command above, the full user name would be **user1@domain.com**.

### A.2. VIEWING USERS IN A SASL DATABASE

To view the user names stored in the SASL database:

```
$ sudo sasldblistusers2 -f qdrouterd.sasldb
user2@domain.com: PASSWORD
user1@domain.com: PASSWORD
```

### A.3. CONFIGURING A SASL DATABASE

To use the SASL database to provide authentication in AMQ Interconnect:

1. Open the **/etc/sasl2/qdrouterd.conf** configuration file.
2. Set the following attributes:

```
pwcheck_method: auxprop
auxprop_plugin: sasl
sasldb_path: SASL_DATABASE_NAME
mech_list: MECHANISM1 ...
```

#### **sasldb\_path**

The name of the SASL database to use.  
For example:

```
sasldb_path: qdrouterd.sasldb
```

#### **mech\_list**

The SASL mechanisms to enable for authentication. To add multiple mechanisms, separate each entry with a space.

For example:

```
mech_list: ANONYMOUS DIGEST-MD5 EXTERNAL PLAIN
```

## APPENDIX B. USING YOUR SUBSCRIPTION

AMQ is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

### Accessing your account

1. Go to [access.redhat.com](https://access.redhat.com).
2. If you do not already have an account, create one.
3. Log in to your account.

### Activating a subscription

1. Go to [access.redhat.com](https://access.redhat.com).
2. Navigate to **My Subscriptions**.
3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

### Downloading zip and tar files

To access zip or tar files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at [access.redhat.com/downloads](https://access.redhat.com/downloads).
2. Locate the **Red Hat AMQ** entries in the **JBOSS INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

### Registering your system for packages

To install RPM packages on Red Hat Enterprise Linux, your system must be registered. If you are using zip or tar files, this step is not required.

1. Go to [access.redhat.com](https://access.redhat.com).
2. Navigate to **Registration Assistant**.
3. Select your OS version and continue to the next page.
4. Use the listed command in your system terminal to complete the registration.

To learn more see [How to Register and Subscribe a System to the Red Hat Customer Portal](#).

*Revised on 2018-08-15 14:52:35 EDT*