Red Hat AMQ 7.6 Using the AMQ Ruby Client

For Use with AMQ Clients 2.7
Abstract

This guide describes how to install and configure the client, run hands-on examples, and use your client with other AMQ components.
# Table of Contents

**CHAPTER 1. OVERVIEW** ................................................................. 4
  1.1. KEY FEATURES ......................................................... 4
  1.2. SUPPORTED STANDARDS AND PROTOCOLS ......................... 4
  1.3. SUPPORTED CONFIGURATIONS .................................. 4
  1.4. TERMS AND CONCEPTS .............................................. 5
  1.5. DOCUMENT CONVENTIONS ...................................... 6
    The sudo command .................................................. 6
    File paths ...................................................... 6
    Variable text ................................................... 6

**CHAPTER 2. INSTALLATION** ..................................................... 7
  2.1. PREREQUISITES .................................................... 7
  2.2. INSTALLING ON RED HAT ENTERPRISE LINUX .................. 7

**CHAPTER 3. GETTING STARTED** ................................................ 8
  3.1. PREREQUISITES .................................................... 8
  3.2. RUNNING HELLO WORLD ON RED HAT ENTERPRISE LINUX .... 8

**CHAPTER 4. EXAMPLES** ........................................................... 9
  4.1. SENDING MESSAGES ............................................... 9
    Running the example ............................................ 9
  4.2. RECEIVING MESSAGES .......................................... 10
    Running the example ........................................... 11

**CHAPTER 5. NETWORK CONNECTIONS** ......................................... 12
  5.1. CONNECTION URLS ................................................. 12

**CHAPTER 6. SENDERS AND RECEIVERS** ....................................... 13
  6.1. CREATING QUEUES AND TOPICS ON DEMAND .................... 13
  6.2. CREATING DURABLE SUBSCRIPTIONS .......................... 13
  6.3. CREATING SHARED SUBSCRIPTIONS ............................ 13

**CHAPTER 7. LOGGING** ........................................................... 14
  7.1. ENABLING PROTOCOL LOGGING .................................. 14

**CHAPTER 8. INTEROPERABILITY** ............................................... 15
  8.1. INTEROPERATING WITH OTHER AMQP CLIENTS .................. 15
  8.2. INTEROPERATING WITH AMQ JMS ............................. 18
    JMS message types ............................................. 18
  8.3. CONNECTING TO AMQ BROKER .................................. 18
  8.4. CONNECTING TO AMQ INTERCONNECT .......................... 19

**APPENDIX A. USING YOUR SUBSCRIPTION** .................................. 20
  A.1. ACCESSING YOUR ACCOUNT .................................... 20
  A.2. ACTIVATING A SUBSCRIPTION .................................. 20
  A.3. DOWNLOADING RELEASE FILES ............................... 20
  A.4. REGISTERING YOUR SYSTEM FOR PACKAGES ................. 20

**APPENDIX B. USING RED HAT ENTERPRISE LINUX PACKAGES** .......... 22
  B.1. OVERVIEW ....................................................... 22
  B.2. SEARCHING FOR PACKAGES ................................... 22
  B.3. INSTALLING PACKAGES ....................................... 22
  B.4. QUERYING PACKAGE INFORMATION ............................ 22
CHAPTER 1. OVERVIEW

AMQ Ruby is a library for developing messaging applications. It enables you to write Ruby applications that send and receive AMQP messages.

IMPORTANT

The AMQ Ruby client is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

AMQ Ruby is part of AMQ Clients, a suite of messaging libraries supporting multiple languages and platforms. For an overview of the clients, see AMQ Clients Overview. For information about this release, see AMQ Clients 2.7 Release Notes.

AMQ Ruby is based on the Proton API from Apache Qpid. For detailed API documentation, see the AMQ Ruby API reference.

1.1. KEY FEATURES

- An event-driven API that simplifies integration with existing applications
- SSL/TLS for secure communication
- Flexible SASL authentication
- Automatic reconnect and failover
- Seamless conversion between AMQP and language-native data types
- Access to all the features and capabilities of AMQP 1.0

1.2. SUPPORTED STANDARDS AND PROTOCOLS

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

- Version 1.0 of the Advanced Message Queueing Protocol (AMQP)
- Versions 1.0, 1.1, 1.2, and 1.3 of the Transport Layer Security (TLS) protocol, the successor to SSL
- Simple Authentication and Security Layer (SASL) mechanisms supported by Cyrus SASL, including ANONYMOUS, PLAIN, SCRAM, EXTERNAL, and GSSAPI (Kerberos)
- Modern TCP with IPv6

1.3. SUPPORTED CONFIGURATIONS

AMQ Ruby supports the OS and language versions listed below. For more information, see Red Hat AMQ 7 Supported Configurations.
- Red Hat Enterprise Linux 7 with Ruby 2.0
- Red Hat Enterprise Linux 8 with Ruby 2.5

AMQ Ruby is supported in combination with the following AMQ components and versions:

- All versions of AMQ Broker
- All versions of AMQ Interconnect
- All versions of AMQ Online
- A-MQ 6 versions 6.2.1 and newer

### 1.4. TERMS AND CONCEPTS

This section introduces the core API entities and describes how they operate together.

**Table 1.1. API terms**

<table>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>A top-level container of connections.</td>
</tr>
<tr>
<td>Connection</td>
<td>A channel for communication between two peers on a network. It contains sessions.</td>
</tr>
<tr>
<td>Session</td>
<td>A context for sending and receiving messages. It contains senders and receivers.</td>
</tr>
<tr>
<td>Sender</td>
<td>A channel for sending messages to a target. It has a target.</td>
</tr>
<tr>
<td>Receiver</td>
<td>A channel for receiving messages from a source. It has a source.</td>
</tr>
<tr>
<td>Source</td>
<td>A named point of origin for messages.</td>
</tr>
<tr>
<td>Target</td>
<td>A named destination for messages.</td>
</tr>
<tr>
<td>Message</td>
<td>An application-specific piece of information.</td>
</tr>
<tr>
<td>Delivery</td>
<td>A message transfer.</td>
</tr>
</tbody>
</table>

AMQ Ruby sends and receives messages. Messages are transferred between connected peers over senders and receivers. Senders and receivers are established over sessions. Sessions are established over connections. Connections are established between two uniquely identified containers. Though a connection can have multiple sessions, often this is not needed. The API allows you to ignore sessions unless you require them.

A sending peer creates a sender to send messages. The sender has a target that identifies a queue or topic at the remote peer. A receiving peer creates a receiver to receive messages. The receiver has a source that identifies a queue or topic at the remote peer.
The sending of a message is called a delivery. The message is the content sent, including all metadata such as headers and annotations. The delivery is the protocol exchange associated with the transfer of that content.

To indicate that a delivery is complete, either the sender or the receiver settles it. When the other side learns that it has been settled, it will no longer communicate about that delivery. The receiver can also indicate whether it accepts or rejects the message.

1.5. DOCUMENT CONVENTIONS

The sudo command
In this document, sudo is used for any command that requires root privileges. Exercise caution when using sudo because any changes can affect the entire system. For more information about sudo, see Using the sudo command.

File paths
In this document, all file paths are valid for Linux, UNIX, and similar operating systems (for example, /home/andrea). On Microsoft Windows, you must use the equivalent Windows paths (for example, C:\Users\andrea).

Variable text
This document contains code blocks with variables that you must replace with values specific to your environment. Variable text is enclosed in arrow braces and styled as italic monospace. For example, in the following command, replace <project-dir> with the value for your environment:

```bash
$ cd <project-dir>
```
CHAPTER 2. INSTALLATION

This chapter guides you through the steps to install AMQ Ruby in your environment.

2.1. PREREQUISITES

- You must have a subscription to access AMQ release files and repositories.
- To install packages on Red Hat Enterprise Linux, you must register your system.
- To use AMQ Ruby, you must install Ruby in your environment.

2.2. INSTALLING ON RED HAT ENTERPRISE LINUX

Procedure

1. Use the subscription-manager command to subscribe to the required package repositories. If necessary, replace `<variant>` with the value for your variant of Red Hat Enterprise Linux (for example, server or workstation).

   Red Hat Enterprise Linux 7
   
   $ sudo subscription-manager repos --enable=amq-clients-2-for-rhel-7-<variant>-rpms

   Red Hat Enterprise Linux 8
   
   $ sudo subscription-manager repos --enable=amq-clients-2-for-rhel-8-x86_64-rpms

2. Use the yum command to install the rubygem-qpid_proton and rubygem-qpid_proton-doc packages.

   $ sudo yum install rubygem-qpid_proton rubygem-qpid_proton-doc

For more information about using packages, see Appendix B, Using Red Hat Enterprise Linux packages.
CHAPTER 3. GETTING STARTED

This chapter guides you through the steps to set up your environment and run a simple messaging program.

3.1. PREREQUISITES

- You must complete the installation procedure for your environment.

- You must have an AMQP 1.0 message broker listening for connections on interface localhost and port 5672. It must have anonymous access enabled. For more information, see Starting the broker.

- You must have a queue named examples. For more information, see Creating a queue.

3.2. RUNNING HELLO WORLD ON RED HAT ENTERPRISE LINUX

The Hello World example creates a connection to the broker, sends a message containing a greeting to the examples queue, and receives it back. On success, it prints the received message to the console.

Change to the examples directory and run the helloworld.rb example.

```
$ cd /usr/share/proton/examples/ruby/
$ ruby helloworld.rb amqp://127.0.0.1 examples
Hello World!
```
CHAPTER 4. EXAMPLES

This chapter demonstrates the use of AMQ Ruby through example programs.

For more examples, see the AMQ Ruby example suite.

4.1. SENDING MESSAGES

This client program connects to a server using `<connection-url>`, creates a sender for target `<address>`, sends a message containing `<message-body>`, closes the connection, and exits.

Example: Sending messages

```ruby
require 'qpid_proton'

class SendHandler < Qpid::Proton::MessagingHandler
  def initialize(conn_url, address, message_body)
    super()
    @conn_url = conn_url
    @address = address
    @message_body = message_body
  end

  def on_container_start(container)
    conn = container.connect(@conn_url)
    conn.open_sender(@address)
  end

  def on_sender_open(sender)
    puts "SEND: Opened sender for target address '#{sender.target.address}'\n"
  end

  def on_sendable(sender)
    message = Qpid::Proton::Message.new(@message_body)
    sender.send(message)
    puts "SEND: Sent message '#{message.body}'\n"
    sender.close
    sender.connection.close
  end
end

if ARGV.size == 3
  conn_url, address, message_body = ARGV
else
  abort "Usage: send.rb <connection-url> <address> <message-body>\n"
end

handler = SendHandler.new(conn_url, address, message_body)
container = Qpid::Proton::Container.new(handler)
container.run
```

Running the example
To run the example program, copy it to a local file and invoke it using the ruby command.

```
$ ruby send.rb amqp://localhost queue1 hello
```

4.2. RECEIVING MESSAGES

This client program connects to a server using <connection-url>, creates a receiver for source <address>, and receives messages until it is terminated or it reaches <count> messages.

Example: Receiving messages

```ruby
require 'qpid_proton'

class ReceiveHandler < Qpid::Proton::MessagingHandler
  def initialize(conn_url, address, desired)
    super()
    @conn_url = conn_url
    @address = address
    @desired = desired
    @received = 0
  end

  def on_container_start(container)
    conn = container.connect(@conn_url)
    conn.open_receiver(@address)
  end

  def on_receiver_open(receiver)
    puts "RECEIVE: Opened receiver for source address '#{receiver.source.address}'"
  end

  def on_message(delivery, message)
    puts "RECEIVE: Received message '#{message.body}'"
    @received += 1

    if @received == @desired
      delivery.receiver.close
      delivery.receiver.connection.close
    end
  end
end

if ARGV.size > 1
  conn_url, address = ARGV[0..1]
else
  abort "Usage: receive.rb <connection-url> <address> [<message-count>]"
end

begin
  desired = Integer(ARGV[2])
rescue TypeError
  desired = 0
```

Red Hat AMQ 7.6 Using the AMQ Ruby Client
end

handler = ReceiveHandler.new(conn_url, address, desired)
container = Qpid::Proton::Container.new(handler)
container.run

Running the example
To run the example program, copy it to a local file and invoke it using the `ruby` command.

$ ruby receive.rb amqp://localhost queue1
5.1. CONNECTION URLS

Connection URLs encode the information used to establish new connections.

**Connection URL syntax**

```
scheme://host[:port]
```

- **Scheme** - The connection transport, either `amqp` for unencrypted TCP or `amqps` for TCP with SSL/TLS encryption.
- **Host** - The remote network host. The value can be a hostname or a numeric IP address. IPv6 addresses must be enclosed in square brackets.
- **Port** - The remote network port. This value is optional. The default value is 5672 for the `amqp` scheme and 5671 for the `amqps` scheme.

**Connection URL examples**

```
amqps://example.com
amqps://example.net:56720
amqp://127.0.0.1
amqp://[::1]:2000
```
CHAPTER 6. SENDERS AND RECEIVERS

The client uses sender and receiver links to represent channels for delivering messages. Senders and receivers are unidirectional, with a source end for the message origin, and a target end for the message destination.

Source and targets often point to queues or topics on a message broker. Sources are also used to represent subscriptions.

6.1. CREATING QUEUES AND TOPICS ON DEMAND

Some message servers support on-demand creation of queues and topics. When a sender or receiver is attached, the server uses the sender target address or the receiver source address to create a queue or topic with a name matching the address.

The message server typically defaults to creating either a queue (for one-to-one message delivery) or a topic (for one-to-many message delivery). The client can indicate which it prefers by setting the `queue` or `topic` capability on the source or target.

For more details, see the following examples:

- `queue-send.rb`
- `queue-receive.rb`
- `topic-send.rb`
- `topic-receive.rb`

6.2. CREATING DURABLE SUBSCRIPTIONS

A durable subscription is a piece of state on the remote server representing a message receiver. Ordinarily, message receivers are discarded when a client closes. However, because durable subscriptions are persistent, clients can detach from them and then re-attach later. Any messages received while detached are available when the client re-attaches.

Durable subscriptions are uniquely identified by combining the client container ID and receiver name to form a subscription ID. These must have stable values so that the subscription can be recovered.

Example

6.3. CREATING SHARED SUBSCRIPTIONS

A shared subscription is a piece of state on the remote server representing one or more message receivers. Because it is shared, multiple clients can consume from the same stream of messages.

The client configures a shared subscription by setting the `shared` capability on the receiver source.

Shared subscriptions are uniquely identified by combining the client container ID and receiver name to form a subscription ID. These must have stable values so that multiple client processes can locate the same subscription. If the `global` capability is set in addition to `shared`, the receiver name alone is used to identify the subscription.

Example
CHAPTER 7. LOGGING

7.1. ENABLING PROTOCOL LOGGING

The client can log AMQP protocol frames to the console. This data is often critical when diagnosing problems.

To enable protocol logging, set the `PN_TRACE_FRM` environment variable to 1:

Example: Enabling protocol logging

```bash
$ export PN_TRACE_FRM=1
$ <your-client-program>
```

To disable protocol logging, unset the `PN_TRACE_FRM` environment variable.
CHAPTER 8. INTEROPERABILITY

This chapter discusses how to use AMQ Ruby in combination with other AMQ components. For an overview of the compatibility of AMQ components, see the product introduction.

8.1. INTEROPERATING WITH OTHER AMQP CLIENTS

AMQP messages are composed using the AMQP type system. This common format is one of the reasons AMQP clients in different languages are able to interoperate with each other.

When sending messages, AMQ Ruby automatically converts language-native types to AMQP-encoded data. When receiving messages, the reverse conversion takes place.

NOTE

More information about AMQP types is available at the interactive type reference maintained by the Apache Qpid project.

Table 8.1. AMQP types

<table>
<thead>
<tr>
<th>AMQP type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>An empty value</td>
</tr>
<tr>
<td>boolean</td>
<td>A true or false value</td>
</tr>
<tr>
<td>char</td>
<td>A single Unicode character</td>
</tr>
<tr>
<td>string</td>
<td>A sequence of Unicode characters</td>
</tr>
<tr>
<td>binary</td>
<td>A sequence of bytes</td>
</tr>
<tr>
<td>byte</td>
<td>A signed 8-bit integer</td>
</tr>
<tr>
<td>short</td>
<td>A signed 16-bit integer</td>
</tr>
<tr>
<td>int</td>
<td>A signed 32-bit integer</td>
</tr>
<tr>
<td>long</td>
<td>A signed 64-bit integer</td>
</tr>
<tr>
<td>ubyte</td>
<td>An unsigned 8-bit integer</td>
</tr>
<tr>
<td>ushort</td>
<td>An unsigned 16-bit integer</td>
</tr>
<tr>
<td>uint</td>
<td>An unsigned 32-bit integer</td>
</tr>
<tr>
<td>ulong</td>
<td>An unsigned 64-bit integer</td>
</tr>
<tr>
<td>float</td>
<td>A 32-bit floating point number</td>
</tr>
<tr>
<td>AMQP type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>double</td>
<td>A 64-bit floating point number</td>
</tr>
<tr>
<td>array</td>
<td>A sequence of values of a single type</td>
</tr>
<tr>
<td>list</td>
<td>A sequence of values of variable type</td>
</tr>
<tr>
<td>map</td>
<td>A mapping from distinct keys to values</td>
</tr>
<tr>
<td>uuid</td>
<td>A universally unique identifier</td>
</tr>
<tr>
<td>symbol</td>
<td>A 7-bit ASCII string from a constrained domain</td>
</tr>
<tr>
<td>timestamp</td>
<td>An absolute point in time</td>
</tr>
</tbody>
</table>

**Table 8.2. AMQ Ruby types before encoding and after decoding**

<table>
<thead>
<tr>
<th>AMQP type</th>
<th>AMQ Ruby type before encoding</th>
<th>AMQ Ruby type after decoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>boolean</td>
<td>true, false</td>
<td>true, false</td>
</tr>
<tr>
<td>char</td>
<td>-</td>
<td>String</td>
</tr>
<tr>
<td>string</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>binary</td>
<td>-</td>
<td>String</td>
</tr>
<tr>
<td>byte</td>
<td>-</td>
<td>Integer</td>
</tr>
<tr>
<td>short</td>
<td>-</td>
<td>Integer</td>
</tr>
<tr>
<td>int</td>
<td>-</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>ubyte</td>
<td>-</td>
<td>Integer</td>
</tr>
<tr>
<td>ushort</td>
<td>-</td>
<td>Integer</td>
</tr>
<tr>
<td>uint</td>
<td>-</td>
<td>Integer</td>
</tr>
<tr>
<td>ulong</td>
<td>-</td>
<td>Integer</td>
</tr>
<tr>
<td>AMQP type</td>
<td>AMQ Ruby type before encoding</td>
<td>AMQ Ruby type after decoding</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>float</td>
<td>-</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Float</td>
<td>Float</td>
</tr>
<tr>
<td>array</td>
<td>-</td>
<td>Array</td>
</tr>
<tr>
<td>list</td>
<td>Array</td>
<td>Array</td>
</tr>
<tr>
<td>map</td>
<td>Hash</td>
<td>Hash</td>
</tr>
<tr>
<td>symbol</td>
<td>Symbol</td>
<td>Symbol</td>
</tr>
<tr>
<td>timestamp</td>
<td>Date, Time</td>
<td>Time</td>
</tr>
</tbody>
</table>

Table 8.3. AMQ Ruby and other AMQ client types (1 of 2)

<table>
<thead>
<tr>
<th>AMQ Ruby type before encoding</th>
<th>AMQ C++ type</th>
<th>AMQ JavaScript type</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>nullptr</td>
<td>null</td>
</tr>
<tr>
<td>true, false</td>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td>String</td>
<td>std::string</td>
<td>string</td>
</tr>
<tr>
<td>Integer</td>
<td>int64_t</td>
<td>number</td>
</tr>
<tr>
<td>Float</td>
<td>double</td>
<td>number</td>
</tr>
<tr>
<td>Array</td>
<td>std::vector</td>
<td>Array</td>
</tr>
<tr>
<td>Hash</td>
<td>std::map</td>
<td>object</td>
</tr>
<tr>
<td>Symbol</td>
<td>proton::symbol</td>
<td>string</td>
</tr>
<tr>
<td>Date, Time</td>
<td>proton::timestamp</td>
<td>number</td>
</tr>
</tbody>
</table>

Table 8.4. AMQ Ruby and other AMQ client types (2 of 2)

<table>
<thead>
<tr>
<th>AMQ Ruby type before encoding</th>
<th>AMQ .NET type</th>
<th>AMQ Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>null</td>
<td>None</td>
</tr>
<tr>
<td>true, false</td>
<td>System.Boolean</td>
<td>bool</td>
</tr>
<tr>
<td>AMQ Ruby type before encoding</td>
<td>AMQ .NET type</td>
<td>AMQ Python type</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>String</td>
<td>System.String</td>
<td>unicode</td>
</tr>
<tr>
<td>Integer</td>
<td>System.Int64</td>
<td>long</td>
</tr>
<tr>
<td>Float</td>
<td>System.Double</td>
<td>float</td>
</tr>
<tr>
<td>Array</td>
<td>Amqp.List</td>
<td>list</td>
</tr>
<tr>
<td>Hash</td>
<td>Amqp.Map</td>
<td>dict</td>
</tr>
<tr>
<td>Symbol</td>
<td>Amqp.Symbol</td>
<td>str</td>
</tr>
<tr>
<td>Date, Time</td>
<td>System.DateTime</td>
<td>long</td>
</tr>
</tbody>
</table>

### 8.2. INTEROPERATING WITH AMQ JMS

AMQP defines a standard mapping to the JMS messaging model. This section discusses the various aspects of that mapping. For more information, see the AMQ JMS Interoperability chapter.

**JMS message types**

AMQ Ruby provides a single message type whose body type can vary. By contrast, the JMS API uses different message types to represent different kinds of data. The table below indicates how particular body types map to JMS message types.

For more explicit control of the resulting JMS message type, you can set the `x-opt-jms-msg-type` message annotation. See the AMQ JMS Interoperability chapter for more information.

#### Table 8.5. AMQ Ruby and JMS message types

<table>
<thead>
<tr>
<th>AMQ Ruby body type</th>
<th>JMS message type</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>TextMessage</td>
</tr>
<tr>
<td>nil</td>
<td>TextMessage</td>
</tr>
<tr>
<td>-</td>
<td>BytesMessage</td>
</tr>
<tr>
<td>Any other type</td>
<td>ObjectMessage</td>
</tr>
</tbody>
</table>

### 8.3. CONNECTING TO AMQ BROKER

AMQ Broker is designed to interoperate with AMQP 1.0 clients. Check the following to ensure the broker is configured for AMQP messaging:

- Port 5672 in the network firewall is open.
- The AMQ Broker AMQP acceptor is enabled. See Default acceptor settings.
- The necessary addresses are configured on the broker. See Addresses, Queues, and Topics.
- The broker is configured to permit access from your client, and the client is configured to send the required credentials. See Broker Security.

8.4. CONNECTING TO AMQ INTERCONNECT

AMQ Interconnect works with any AMQP 1.0 client. Check the following to ensure the components are configured correctly:

- Port 5672 in the network firewall is open.
- The router is configured to permit access from your client, and the client is configured to send the required credentials. See Securing network connections.
APPENDIX A. USING YOUR SUBSCRIPTION

AMQ is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

A.1. ACCESSING YOUR ACCOUNT

Procedure

1. Go to access.redhat.com.
2. If you do not already have an account, create one.
3. Log in to your account.

A.2. ACTIVATING A SUBSCRIPTION

Procedure

1. Go to access.redhat.com.
2. Navigate to My Subscriptions.
3. Navigate to Activate a subscription and enter your 16-digit activation number.

A.3. DOWNLOADING RELEASE FILES

To access .zip, .tar.gz, and other release files, use the customer portal to find the relevant files for download. If you are using RPM packages or the Red Hat Maven repository, this step is not required.

Procedure

1. Open a browser and log in to the Red Hat Customer Portal Product Downloads page at access.redhat.com/downloads.
2. Locate the Red Hat AMQ entries in the INTEGRATION AND AUTOMATION category.
3. Select the desired AMQ product. The Software Downloads page opens.
4. Click the Download link for your component.

A.4. REGISTERING YOUR SYSTEM FOR PACKAGES

To install RPM packages on Red Hat Enterprise Linux, your system must be registered. If you are using downloaded release files, this step is not required.

Procedure

1. Go to 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.

For more information, see How to Register and Subscribe a System to the Red Hat Customer Portal.
APPENDIX B. USING RED HAT ENTERPRISE LINUX PACKAGES

This section describes how to use software delivered as RPM packages for Red Hat Enterprise Linux.

B.1. OVERVIEW

A component such as a library or server often has multiple packages associated with it. You do not have to install them all. You can install only the ones you need.

The primary package typically has the simplest name, without additional qualifiers. This package provides all the required interfaces for using the component at program run time.

Packages with names ending in `-devel` contain headers for C and C++ libraries. These are required at compile time to build programs that depend on this package.

Packages with names ending in `-docs` contain documentation and example programs for the component.

For more information about using RPM packages, see one of the following resources:

- Red Hat Enterprise Linux 6 - Installing and managing software
- Red Hat Enterprise Linux 7 - Installing and managing software
- Red Hat Enterprise Linux 8 - Managing software packages

B.2. SEARCHING FOR PACKAGES

To search for packages, use the `yum search` command. The search results include package names, which you can use as the value for `<package>` in the other commands listed in this section.

```
$ yum search <keyword>...
```

B.3. INSTALLING PACKAGES

To install packages, use the `yum install` command.

```
$ sudo yum install <package>...
```

B.4. QUERYING PACKAGE INFORMATION

To list the packages installed in your system, use the `rpm -qa` command.

```
$ rpm -qa
```

To get information about a particular package, use the `rpm -qi` command.

```
$ rpm -qi <package>
```

To list all the files associated with a package, use the `rpm -ql` command.

```
$ rpm -ql <package>
```
APPENDIX C. USING AMQ BROKER WITH THE EXAMPLES

The AMQ Ruby examples require a running message broker with a queue named examples. Use the procedures below to install and start the broker and define the queue.

C.1. INSTALLING THE BROKER

Follow the instructions in Getting Started with AMQ Broker to install the broker and create a broker instance. Enable anonymous access.

The following procedures refer to the location of the broker instance as <broker-instance-dir>.

C.2. STARTING THE BROKER

Procedure

1. Use the artemis run command to start the broker.

   $ <broker-instance-dir>/bin/artemis run

2. Check the console output for any critical errors logged during startup. The broker logs Server is now live when it is ready.

   $ example-broker/bin/artemis run

   Red Hat AMQ <version>

   2020-06-03 12:12:11,807 INFO  [org.apache.activemq.artemis.integration.bootstrap] AMQ101000: Starting ActiveMQ Artemis Server ...

   2020-06-03 12:12:12,336 INFO  [org.apache.activemq.artemis.core.server] AMQ221007: Server is now live ...

C.3. CREATING A QUEUE

In a new terminal, use the artemis queue command to create a queue named examples.

   $ <broker-instance-dir>/bin/artemis queue create --name examples --address examples --auto-create-address --anycast

   You are prompted to answer a series of yes or no questions. Answer N for no to all of them.

   Once the queue is created, the broker is ready for use with the example programs.

C.4. STOPPING THE BROKER

24
When you are done running the examples, use the `artemis stop` command to stop the broker.

```
$ <broker-instance-dir>/bin/artemis stop
```

Revised on 2020-06-16 17:53:38 UTC