Red Hat AMQ 7.2

Using the AMQ Ruby Client

For Use with AMQ Clients 2.3

Last Updated: 2019-03-18

Red Hat AMQ 7.2 Using the AMQ Ruby Client

For Use with AMQ Clients 2.3

Legal Notice

Copyright © 2019 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 and configure the client, run hands-on examples, and use your client with other AMQ components.

Table of Contents

CHAPTER 1. OVERVIEW **3**
1.1. KEY FEATURES 3
1.2. SUPPORTED STANDARDS AND PROTOCOLS 3
1.3. SUPPORTED CONFIGURATIONS 3
1.4. TERMS AND CONCEPTS 4
1.5. DOCUMENT CONVENTIONS 4
The sudo command 4
About the use of file paths in this document 5

CHAPTER 2. INSTALLATION **6**
2.1. PREREQUISITES 6
2.2. INSTALLING ON RED HAT ENTERPRISE LINUX 6

CHAPTER 3. GETTING STARTED **7**
3.1. PREPARING THE BROKER 7
3.2. RUNNING HELLO WORLD 7

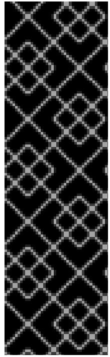
CHAPTER 4. EXAMPLES **8**
4.1. SENDING MESSAGES 8
Running the example 8
4.2. RECEIVING MESSAGES 9
Running the example 10

CHAPTER 5. INTEROPERABILITY **11**
5.1. INTEROPERATING WITH OTHER AMQP CLIENTS 11
5.2. INTEROPERATING WITH AMQ JMS 14
JMS message types 14
5.3. CONNECTING TO AMQ BROKER 14
5.4. CONNECTING TO AMQ INTERCONNECT 15

APPENDIX A. USING YOUR SUBSCRIPTION **16**
Accessing your account 16
Activating a subscription 16
Downloading zip and tar files 16
Registering your system for packages 16

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 2.3 Release Notes.

AMQ Ruby is based on the Proton API from Apache Qpid.

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, and 1.2 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 is supported on Red Hat Enterprise Linux 7 with Ruby 2.0.

For more information, see Red Hat AMQ 7 Supported Configurations.

1.4. TERMS AND CONCEPTS

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

Table 1.1. API terms

Entity	Description
Container	A top-level container of connections
Connection	A channel for communication between two peers on a network
Session	A context for sending and receiving messages
Sender	A channel for sending messages to a target
Receiver	A channel for receiving messages from a source
Source	A named point of origin for messages
Target	A named destination for messages
Message	A mutable holder of application data
Delivery	A message transfer

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

This document uses the following conventions for the **sudo** command and file paths.

The sudo command

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.

About the use of file paths in this document

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

CHAPTER 2. INSTALLATION

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

2.1. PREREQUISITES

To begin installation, use your subscription to access AMQ distribution files and repositories.

2.2. INSTALLING ON RED HAT ENTERPRISE LINUX

AMQ Ruby is distributed as a set of RPM packages for Red Hat Enterprise Linux. Follow these steps to install them.

1. Use the **subscription-manager** command to subscribe to the required package repositories.

   Red Hat Enterprise Linux 7

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

CHAPTER 3. GETTING STARTED

This chapter guides you through a simple exercise to help you get started using AMQ Ruby.

3.1. PREPARING THE BROKER

The example programs require a running broker with a queue named **`examples`**. Follow these steps to define the queue and start the broker:

Procedure

1. Install the broker.
2. Create a broker instance. Enable anonymous access.
3. Start the broker instance and check the console for any critical errors logged during startup.

   ```
   $ <broker-instance-dir>/bin/artemis run
   ...
   14:43:20,158 INFO
   [org.apache.activemq.artemis.integration.bootstrap] AMQ101000:
   Starting ActiveMQ Artemis Server
   ...
   15:01:39,686 INFO  [org.apache.activemq.artemis.core.server]
   AMQ221020: Started Acceptor at 0.0.0.0:5672 for protocols [AMQP]
   ...
   15:01:39,691 INFO  [org.apache.activemq.artemis.core.server]
   AMQ221007: Server is now live
   ```

4. Use the **`artemis queue`** command to create a queue called **`examples`**.

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

   You are prompted to answer a series of questions. For yes or no questions, type **`N`**. Otherwise, press Enter to accept the default value.

3.2. RUNNING HELLO WORLD

The Hello World example sends a message to the **`examples`** queue on the broker and then fetches it back. On success it prints **`Hello World!`** to the console.

Using a new terminal window, change directory to the AMQ Ruby examples directory and run the **`helloworld.rb`** example.

```
$ cd /usr/share/proton-0.27.0/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.

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

```
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

```
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}'\n"
  end

  def on_message(delivery, message)
    puts "RECEIVE: Received message '#{message.body}'\n"

    @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>]\n"
end

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

```
  desired = 0
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
```

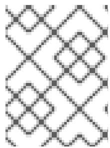
CHAPTER 5. INTEROPERABILITY

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.

5.1. 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 5.1. AMQP types

AMQP type	Description
`null`	An empty value
`boolean`	A true or false value
`char`	A single Unicode character
`string`	A sequence of Unicode characters
`binary`	A sequence of bytes
`byte`	A signed 8-bit integer
`short`	A signed 16-bit integer
`int`	A signed 32-bit integer
`long`	A signed 64-bit integer
`ubyte`	An unsigned 8-bit integer
`ushort`	An unsigned 16-bit integer
`uint`	An unsigned 32-bit integer
`ulong`	An unsigned 64-bit integer
`float`	A 32-bit floating point number

AMQP type	Description
`double`	A 64-bit floating point number
`array`	A sequence of values of a single type
`list`	A sequence of values of variable type
`map`	A mapping from distinct keys to values
`uuid`	A universally unique identifier
`symbol`	A 7-bit ASCII string from a constrained domain
`timestamp`	An absolute point in time

Table 5.2. AMQ Ruby types before encoding and after decoding

AMQP type	AMQ Ruby type before encoding	AMQ Ruby type after decoding
`null`	`nil`	`nil`
`boolean`	`true, false`	`true, false`
`char`	-	`String`
`string`	`String`	`String`
`binary`	-	`String`
`byte`	-	`Integer`
`short`	-	`Integer`
`int`	-	`Integer`
`long`	`Integer`	`Integer`
`ubyte`	-	`Integer`
`ushort`	-	`Integer`
`uint`	-	`Integer`
`ulong`	-	`Integer`

AMQP type	AMQ Ruby type before encoding	AMQ Ruby type after decoding
float	-	Float
double	Float	Float
array	-	Array
list	Array	Array
map	Hash	Hash
symbol	Symbol	Symbol
timestamp	Date, Time	Time

Table 5.3. AMQ Ruby and other AMQ client types (1 of 2)

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

Table 5.4. AMQ Ruby and other AMQ client types (2 of 2)

AMQ Ruby type before encoding	AMQ .NET type	AMQ Python type
nil	null	None
true, false	System.Boolean	bool

AMQ Ruby type before encoding	AMQ .NET type	AMQ Python type
`String`	`System.String`	`unicode`
`Integer`	`System.Int64`	`long`
`Float`	`System.Double`	`float`
`Array`	`Amqp.List`	`list`
`Hash`	`Amqp.Map`	`dict`
`Symbol`	`Amqp.Symbol`	`str`
`Date, Time`	`System.DateTime`	`long`

5.2. 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 5.5. AMQ Ruby and JMS message types

AMQ Ruby body type	JMS message type
`String`	`TextMessage`
`nil`	`TextMessage`
`-`	`BytesMessage`
Any other type	`ObjectMessage`

5.3. 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.

5.4. 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 Interconnect Security.

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.

Accessing your account

1. Go to 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.
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.
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.
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 2019-03-18 15:32:55 UTC