



Red Hat AMQ 7.5

Using the AMQ Python Client

For Use with AMQ Clients 2.6

Red Hat AMQ 7.5 Using the AMQ Python Client

For Use with AMQ Clients 2.6

Legal Notice

Copyright © 2020 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, the Red Hat 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 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	4
1.1. KEY FEATURES	4
1.2. SUPPORTED CONFIGURATIONS	4
1.3. TERMS AND CONCEPTS	5
1.4. DOCUMENT CONVENTIONS	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. PREPARING THE BROKER	8
3.2. RUNNING HELLO WORLD	8
CHAPTER 4. EXAMPLES	9
4.1. SENDING MESSAGES	9
Running the example	10
4.2. RECEIVING MESSAGES	10
Running the example	11
CHAPTER 5. USING THE API	12
5.1. BASIC OPERATION	12
5.1.1. Handling messaging events	12
5.1.2. Creating a container	12
Setting the container identity	12
5.2. NETWORK CONNECTIONS	13
5.2.1. Connection URLs	13
5.2.2. Creating outgoing connections	13
5.2.3. Listening for incoming connections	14
5.2.4. Configuring reconnect	14
5.2.5. Configuring failover	15
5.3. SENDERS AND RECEIVERS	15
5.3.1. Creating queues and topics on demand	15
5.3.2. Creating durable subscriptions	16
5.3.3. Creating shared subscriptions	17
5.4. MESSAGE DELIVERY	17
5.4.1. Sending messages	17
5.4.2. Tracking sent messages	18
5.4.3. Receiving messages	18
5.4.4. Acknowledging received messages	18
5.5. SECURITY	19
5.5.1. Securing connections with SSL/TLS	19
5.5.2. Connecting with a user and password	19
5.5.3. Configuring SASL authentication	19
5.5.4. Authenticating using Kerberos	20
5.6. LOGGING	20
5.6.1. Enabling protocol logging	20
5.7. TRACING	21
CHAPTER 6. FILE-BASED CONFIGURATION	22
6.1. FILE LOCATIONS	22
6.2. FILE FORMAT	22

6.3. CONFIGURATION OPTIONS	23
CHAPTER 7. INTEROPERABILITY	24
7.1. INTEROPERATING WITH OTHER AMQP CLIENTS	24
7.2. INTEROPERATING WITH AMQ JMS	28
JMS message types	28
7.3. CONNECTING TO AMQ BROKER	28
7.4. CONNECTING TO AMQ INTERCONNECT	29
APPENDIX A. USING YOUR SUBSCRIPTION	30
Accessing your account	30
Activating a subscription	30
Downloading ZIP and TAR files	30
Registering your system for packages	30

CHAPTER 1. OVERVIEW

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

AMQ Python 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 2.6 Release Notes](#).

AMQ Python is based on the Proton API from [Apache Qpid](#). For detailed API documentation, see the [AMQ Python 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
- Distributed tracing based on the OpenTracing standard (RHEL 7 and 8)



IMPORTANT

Distributed tracing in AMQ Clients 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/>.

- Seamless conversion between AMQP and language-native data types
- Access to all the features and capabilities of AMQP 1.0

1.2. SUPPORTED CONFIGURATIONS

AMQ Python supports the following OS and language versions:

- Red Hat Enterprise Linux 6 with Python 2.6
- Red Hat Enterprise Linux 7 with Python 2.7
- Red Hat Enterprise Linux 8 with Python 3.6
- Microsoft Windows 10 Pro with Python 2.7
- Microsoft Windows Server 2012 R2 and 2016 with Python 2.7

AMQ Python 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 higher

For more information, see [Red Hat AMQ 7 Supported Configurations](#) .

1.3. 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. It contains sessions.
Session	A context for sending and receiving messages. It contains senders and receivers.
Sender	A channel for sending messages to a target. It has a target.
Receiver	A channel for receiving messages from a source. It has a source.
Source	A named point of origin for messages.
Target	A named destination for messages.
Message	An application-specific piece of information.
Delivery	A message transfer.

AMQ Python 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.

1.4. 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

This chapter guides you through the steps to install AMQ Python 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 Python 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 6

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

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 **python-qpiproton** and **python-qpiproton-docs** packages.

```
$ sudo yum install python-qpiproton python-qpiproton-docs
```

CHAPTER 3. GETTING STARTED

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

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 Python examples directory and run the **helloworld.py** example.

```
$ cd /usr/share/proton-0.30.0/examples/python/
$ python helloworld.py
Hello World!
```

CHAPTER 4. EXAMPLES

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

For more examples, see the [AMQ Python 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

```

from __future__ import print_function

import sys

from proton import Message
from proton.handlers import MessagingHandler
from proton.reactor import Container

class SendHandler(MessagingHandler):
    def __init__(self, conn_url, address, message_body):
        super(SendHandler, self).__init__()

        self.conn_url = conn_url
        self.address = address
        self.message_body = message_body

    def on_start(self, event):
        conn = event.container.connect(self.conn_url)

        # To connect with a user and password:
        # conn = event.container.connect(self.conn_url, user="<user>", password="<password>")

        event.container.create_sender(conn, self.address)

    def on_link_opened(self, event):
        print("SEND: Opened sender for target address '{0}'".format
              (event.sender.target.address))

    def on_sendable(self, event):
        message = Message(self.message_body)
        event.sender.send(message)

        print("SEND: Sent message '{0}'".format(message.body))

        event.sender.close()
        event.connection.close()

def main():
    try:
        conn_url, address, message_body = sys.argv[1:4]
    except ValueError:
        sys.exit("Usage: send.py <connection-url> <address> <message-body>")

```

```

handler = SendHandler(conn_url, address, message_body)
container = Container(handler)
container.run()

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        pass

```

Running the example

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

```
$ python send.py 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

```

from __future__ import print_function

import sys

from proton.handlers import MessagingHandler
from proton.reactor import Container

class ReceiveHandler(MessagingHandler):
    def __init__(self, conn_url, address, desired):
        super(ReceiveHandler, self).__init__()

        self.conn_url = conn_url
        self.address = address
        self.desired = desired
        self.received = 0

    def on_start(self, event):
        conn = event.container.connect(self.conn_url)

        # To connect with a user and password:
        # conn = event.container.connect(self.conn_url, user="<user>", password="<password>")

        event.container.create_receiver(conn, self.address)

    def on_link_opened(self, event):
        print("RECEIVE: Created receiver for source address '{0}'".format
              (self.address))

    def on_message(self, event):
        message = event.message

```

```
print("RECEIVE: Received message '{0}'".format(message.body))

self.received += 1

if self.received == self.desired:
    event.receiver.close()
    event.connection.close()

def main():
    try:
        conn_url, address = sys.argv[1:3]
    except ValueError:
        sys.exit("Usage: receive.py <connection-url> <address> [<message-count>]")

    try:
        desired = int(sys.argv[3])
    except (IndexError, ValueError):
        desired = 0

    handler = ReceiveHandler(conn_url, address, desired)
    container = Container(handler)
    container.run()

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        pass
```

Running the example

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

```
$ python receive.py amqp://localhost queue1
```

CHAPTER 5. USING THE API

This chapter explains how to use the AMQ Python API to perform common messaging tasks.

For more information, see the [AMQ Python API reference](#) and [AMQ Python example suite](#).

5.1. BASIC OPERATION

5.1.1. Handling messaging events

AMQ Python is an asynchronous event-driven API. To define how an application handles events, the user implements callback methods on the **MessagingHandler** class. These methods are then called as network activity or timers trigger new events.

Example: Handling messaging events

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        print("The container event loop has started")

    def on_sendable(self, event):
        print("A message can be sent")

    def on_message(self, event):
        print("A message is received")
```

These are only a few common-case events. The full set is documented in the [API reference](#).

The **event** argument has attributes for accessing the object the event is regarding. Attributes with no relevance to a particular event are null.

Example: Accessing event objects

```
event.container
event.connection
event.session
event.sender
event.receiver
event.delivery
event.message
```

5.1.2. Creating a container

The container is the top-level API object. It is the entry point for creating connections, and it is responsible for running the main event loop. It is often constructed with a global event handler.

Example: Creating a container

```
handler = ExampleHandler()
container = Container(handler)
container.run()
```

Setting the container identity

Each container instance has a unique identity called the container ID. When AMQ Python makes a connection, it sends the container ID to the remote peer. To set the container ID, pass it to the **Container** constructor.

Example: Setting the container identity

```
container = Container(handler)
container.container_id = "job-processor-3"
```

If the user does not set the ID, the library will generate a UUID when the container is constructed.

5.2. NETWORK CONNECTIONS

5.2.1. 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
```

5.2.2. Creating outgoing connections

To connect to a remote server, call the **Container.connect()** method with a [connection URL](#). This is typically done inside the **MessagingHandler.on_start()** method.

Example: Creating outgoing connections

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        event.container.connect("amqp://example.com")

    def on_connection_opened(self, event):
        print("Connection", event.connection, "is open")
```

See the [Section 5.5, "Security"](#) section for information about creating secure connections.

5.2.3. Listening for incoming connections

AMQ Python can accept inbound network connections, allowing you to build messaging servers. To start listening for connections, use the **Container.listen()** method with a URL containing the local host address and port to listen on.

Example: Listening for incoming connections

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        event.container.listen("0.0.0.0")

    def on_connection_opened(self, event):
        print("New incoming connection", event.connection)
```

The special IP address **0.0.0.0** listens on all available IPv4 interfaces. To listen on all IPv6 interfaces, use **:::0**.

For more information, see the [server receive.py example](#).

5.2.4. Configuring reconnect

Reconnect allows a client to recover from lost connections. It is used to ensure that the components in a distributed system reestablish communication after temporary network or component failures.

AMQ Python enables reconnect by default. If a connection is lost or a connection attempt fails, the client will try again after a brief delay. The delay increases exponentially for each new attempt, up to a default maximum of 10 seconds.

To disable reconnect, set the **reconnect** connection option to **False**.

Example: Disabling reconnect

```
container.connect("amqp://example.com", reconnect=False)
```

To control the delays between connection attempts, define a class implementing the **reset()** and **next()** methods and set the **reconnect** connection option to an instance of that class.

Example: Configuring reconnect

```
class ExampleReconnect(object):
    def __init__(self):
        self.delay = 0

    def reset(self):
        self.delay = 0

    def next(self):
        if self.delay == 0:
            self.delay = 0.1
        else:
            self.delay = min(10, 2 * self.delay)
```

```
return self.delay
```

```
container.connect("amqp://example.com", reconnect=ExampleReconnect())
```

The **next** method returns the next delay in seconds. The **reset** method is called once before the reconnect process begins.

5.2.5. Configuring failover

AMQ Python allows you to configure multiple connection endpoints. If connecting to one fails, the client attempts to connect to the next in the list. If the list is exhausted, the process starts over.

To specify multiple connection endpoints, set the **urls** connection option to a list of connection URLs.

Example: Configuring failover

```
urls = ["amqp://alpha.example.com", "amqp://beta.example.com"]
container.connect(urls=urls)
```

It is an error to use the **url** and **urls** options at the same time.

5.3. 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.

5.3.1. 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.

To select queue or topic semantics, follow these steps:

1. Configure your message server for automatic creation of queues and topics. This is often the default configuration.
2. Set either the **queue** or **topic** capability on your sender target or receiver source, as in the examples below.

Example: Sending to a queue created on demand

```
class CapabilityOptions(SenderOption):
    def apply(self, sender):
        sender.target.capabilities.put_object(symbol("queue"))
```

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        conn = event.container.connect("amqp://example.com")
        event.container.create_sender(conn, "jobs", options=CapabilityOptions())
```

Example: Receiving from a topic created on demand

```
class CapabilityOptions(ReceiverOption):
    def apply(self, receiver):
        receiver.source.capabilities.put_object(symbol("topic"))

class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        conn = event.container.connect("amqp://example.com")
        event.container.create_receiver(conn, "notifications", options=CapabilityOptions())
```

For more information, see the following examples:

- [queue-send.py](#)
- [queue-receive.py](#)
- [topic-send.py](#)
- [topic-receive.py](#)

5.3.2. 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.

To create a durable subscription, follow these steps:

1. Set the connection container ID to a stable value, such as **client-1**:

```
container = Container(handler)
container.container_id = "client-1"
```

2. Configure the receiver source for durability by setting the **durability** and **expiry_policy** properties:

```
class SubscriptionOptions(ReceiverOption):
    def apply(self, receiver):
        receiver.source.durability = Terminus.DELIVERIES
        receiver.source.expiry_policy = Terminus.EXPIRE_NEVER
```

3. Create a receiver with a stable name, such as **sub-1**, and apply the source properties:

```
event.container.create_receiver(conn, "notifications",
                               name="sub-1",
                               options=SubscriptionOptions())
```

To detach from a subscription, use the **Receiver.detach()** method. To terminate the subscription, use the **Receiver.close()** method.

For more information, see the [durable-subscribe.py example](#).

5.3.3. 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.

To create a durable subscription, follow these steps:

1. Set the connection container ID to a stable value, such as **client-1**:

```
container = Container(handler)
container.container_id = "client-1"
```

2. Configure the receiver source for sharing by setting the **shared** capability:

```
class SubscriptionOptions(ReceiverOption):
    def apply(self, receiver):
        receiver.source.capabilities.put_object(symbol("shared"))
```

3. Create a receiver with a stable name, such as **sub-1**, and apply the source properties:

```
event.container.create_receiver(conn, "notifications",
                               name="sub-1",
                               options=SubscriptionOptions())
```

To detach from a subscription, use the **Receiver.detach()** method. To terminate the subscription, use the **Receiver.close()** method.

For more information, see the [shared-subscribe.py example](#).

5.4. MESSAGE DELIVERY

5.4.1. Sending messages

To send a message, override the **on_sendable** event handler and call the **Sender.send()** method. The **sendable** event fires when the **Sender** has enough credit to send at least one message.

Example: Sending messages

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        conn = event.container.connect("amqp://example.com")
        sender = event.container.create_sender(conn, "jobs")

    def on_sendable(self, event):
        message = Message("job-content")
        event.sender.send(message)
```

For more information, see the [send.py example](#).

5.4.2. Tracking sent messages

When a message is sent, the sender can keep a reference to the **delivery** object representing the transfer. After the message is delivered, the receiver accepts or rejects it. The sender is notified of the outcome for each delivery.

To monitor the outcome of a sent message, override the **on_accepted** and **on_rejected** event handlers and map the delivery state update to the delivery returned from **send()**.

Example: Tracking sent messages

```
def on_sendable(self, event):
    message = Message(self.message_body)
    delivery = event.sender.send(message)

def on_accepted(self, event):
    print("Delivery", event.delivery, "is accepted")

def on_rejected(self, event):
    print("Delivery", event.delivery, "is rejected")
```

5.4.3. Receiving messages

To receive a message, create a receiver and override the **on_message** event handler.

Example: Receiving messages

```
class ExampleHandler(MessagingHandler):
    def on_start(self, event):
        conn = event.container.connect("amqp://example.com")
        receiver = event.container.create_receiver(conn, "jobs")

    def on_message(self, event):
        print("Received message", event.message, "from", event.receiver)
```

For more information, see the [receive.py example](#).

5.4.4. Acknowledging received messages

To explicitly accept or reject a delivery, use the **Delivery.update()** method with the **ACCEPTED** or **REJECTED** state in the **on_message** event handler.

Example: Acknowledging received messages

```
def on_message(self, event):
    try:
        process_message(event.message)
        event.delivery.update(ACCEPTED)
    except:
        event.delivery.update(REJECTED)
```

By default, if you do not explicitly acknowledge a delivery, then the library accepts it after `on_message` returns. To disable this behavior, set the `auto_accept` receiver option to false.

5.5. SECURITY

5.5.1. Securing connections with SSL/TLS

AMQ Python uses SSL/TLS to encrypt communication between clients and servers.

To connect to a remote server with SSL/TLS, use a connection URL with the `amqps` scheme.

Example: Enabling SSL/TLS

```
container.connect("amqps://example.com")
```

5.5.2. Connecting with a user and password

AMQ Python can authenticate connections with a user and password.

To specify the credentials used for authentication, set the `user` and `password` options on the `connect()` method.

Example: Connecting with a user and password

```
container.connect("amqps://example.com", user="alice", password="secret")
```

5.5.3. Configuring SASL authentication

AMQ Python uses the SASL protocol to perform authentication. SASL can use a number of different authentication *mechanisms*. When two network peers connect, they exchange their allowed mechanisms, and the strongest mechanism allowed by both is selected.



NOTE

The client uses Cyrus SASL to perform authentication. Cyrus SASL uses plug-ins to support specific SASL mechanisms. Before you can use a particular SASL mechanism, the relevant plug-in must be installed. For example, you need the `cyrus-sasl-plain` plug-in in order to use SASL PLAIN authentication.

To see a list of Cyrus SASL plug-ins in Red Hat Enterprise Linux, use the `yum search cyrus-sasl` command. To install a Cyrus SASL plug-in, use the `yum install PLUG-IN` command.

By default, AMQ Python allows all of the mechanisms supported by the local SASL library configuration. To restrict the allowed mechanisms and thereby control what mechanisms can be negotiated, use the **allowed_mechs** connection option. It takes a string containing a space-separated list of mechanism names.

Example: Configuring SASL authentication

```
container.connect("amqps://example.com", allowed_mechs="ANONYMOUS")
```

This example forces the connection to authenticate using the **ANONYMOUS** mechanism even if the server we connect to offers other options. Valid mechanisms include **ANONYMOUS**, **PLAIN**, **SCRAM-SHA-256**, **SCRAM-SHA-1**, **GSSAPI**, and **EXTERNAL**.

AMQ Python enables SASL by default. To disable it, set the **sasl_enabled** connection option to false.

Example: Disabling SASL

```
event.container.connect("amqps://example.com", sasl_enabled=False)
```

5.5.4. Authenticating using Kerberos

Kerberos is a network protocol for centrally managed authentication based on the exchange of encrypted tickets. See [Using Kerberos](#) for more information.

1. Configure Kerberos in your operating system. See [Configuring Kerberos](#) to set up Kerberos on Red Hat Enterprise Linux.
2. Enable the **GSSAPI** SASL mechanism in your client application.

```
container.connect("amqps://example.com", allowed_mechs="GSSAPI")
```

3. Use the **kinit** command to authenticate your user credentials and store the resulting Kerberos ticket.

```
$ kinit <user>@<realm>
```

4. Run the client program.

5.6. LOGGING

5.6.1. 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

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


To disable protocol logging, unset the **PN_TRACE_FRM** environment variable.

5.7. TRACING

The client offers distributed tracing based on the Jaeger implementation of the OpenTracing standard. Use the following steps to enable tracing in your application:

1. Install the tracing dependencies.

Red Hat Enterprise Linux 7

```
$ sudo yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
$ sudo yum install python2-pip
$ pip install --user --upgrade setuptools
$ pip install --user opentracing jaeger-client
```

Red Hat Enterprise Linux 8

```
$ sudo dnf install python3-pip
$ pip3 install --user opentracing jaeger-client
```

2. Register the global tracer in your program.

Example: Global tracer configuration

```
from proton.tracing import init_tracer

tracer = init_tracer("<service-name>")
```

For more information about Jaeger configuration, see [Jaeger Sampling](#).

When testing or debugging, you may want to force Jaeger to trace a particular operation. See the [Jaeger Python client documentation](#) for more information.

To view the traces your application captures, use the [Jaeger Getting Started](#) to run the Jaeger infrastructure and console.

CHAPTER 6. FILE-BASED CONFIGURATION

AMQ Python can read the configuration options used to establish connections from a local file named **connect.json**. This enables you to configure connections in your application at the time of deployment.

The library attempts to read the file when the application calls the container **connect** method without supplying any connection options.

6.1. FILE LOCATIONS

If set, AMQ Python uses the value of the **MESSAGING_CONNECT_FILE** environment variable to locate the configuration file.

If **MESSAGING_CONNECT_FILE** is not set, AMQ Python searches for a file named **connect.json** at the following locations and in the order shown. It stops at the first match it encounters.

On Linux:

1. **\$PWD/connect.json**, where **\$PWD** is the current working directory of the client process
2. **\$HOME/.config/messaging/connect.json**, where **\$HOME** is the current user home directory
3. **/etc/messaging/connect.json**

On Windows:

1. **%cd%/connect.json**, where **%cd%** is the current working directory of the client process

If no **connect.json** file is found, the library uses default values for all options.

6.2. FILE FORMAT

The **connect.json** file contains JSON data, with additional support for JavaScript comments.

All of the configuration attributes are optional or have default values, so a simple example need only provide a few details:

Example: A simple connect.json file

```
{
  "host": "example.com",
  "user": "alice",
  "password": "secret"
}
```

SASL and SSL/TLS options are nested under **"sasl"** and **"tls"** namespaces:

Example: A connect.json file with SASL and SSL/TLS options

```
{
  "host": "example.com",
  "user": "ortega",
  "password": "secret",
  "sasl": {
```

```

    "mechanisms": ["SCRAM-SHA-1", "SCRAM-SHA-256"]
  },
  "tls": {
    "cert": "/home/ortega/cert.pem",
    "key": "/home/ortega/key.pem"
  }
}

```

6.3. CONFIGURATION OPTIONS

The option keys containing a dot (.) represent attributes nested inside a namespace.

Table 6.1. Configuration options in `connect.json`

Key	Value type	Default value	Description
scheme	string	"amqps"	"amqp" for cleartext or "amqps" for SSL/TLS
host	string	"localhost"	The hostname or IP address of the remote host
port	string or number	"amqps"	A port number or port literal
user	string	<i>None</i>	The user name for authentication
password	string	<i>None</i>	The password for authentication
sasl.mechanisms	list or string	<i>None</i> (<i>system defaults</i>)	A JSON list of enabled SASL mechanisms. A bare string represents one mechanism. If none are specified, the client uses the default mechanisms provided by the system.
sasl.allow_insecure	boolean	false	Enable mechanisms that send cleartext passwords
tls.cert	string	<i>None</i>	The filename or database ID of the client certificate
tls.key	string	<i>None</i>	The filename or database ID of the private key for the client certificate
tls.ca	string	<i>None</i>	The filename, directory, or database ID of the CA certificate
tls.verify	boolean	true	Require a valid server certificate with a matching hostname

CHAPTER 7. INTEROPERABILITY

This chapter discusses how to use AMQ Python in combination with other AMQ components. For an overview of the compatibility of AMQ components, see the [product introduction](#).

7.1. 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 Python 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 7.1. 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 7.2. AMQ Python types before encoding and after decoding

AMQP type	AMQ Python type before encoding	AMQ Python type after decoding
null	None	None
boolean	bool	bool
char	proton.char	unicode
string	unicode	unicode
binary	bytes	bytes
byte	proton.byte	int
short	proton.short	int
int	proton.int32	long
long	long	long
ubyte	proton.ubyte	long
ushort	proton.ushort	long
uint	proton.uint	long
ulong	proton.ulong	long

AMQP type	AMQ Python type before encoding	AMQ Python type after decoding
float	proton.float32	float
double	float	float
array	proton.Array	proton.Array
list	list	list
map	dict	dict
symbol	proton.symbol	str
timestamp	proton.timestamp	long

Table 7.3. AMQ Python and other AMQ client types (1 of 2)

AMQ Python type before encoding	AMQ C++ type	AMQ JavaScript type
None	nullptr	null
bool	bool	boolean
proton.char	wchar_t	number
unicode	std::string	string
bytes	proton::binary	string
proton.byte	int8_t	number
proton.short	int16_t	number
proton.int32	int32_t	number
long	int64_t	number
proton.ubyte	uint8_t	number
proton.ushort	uint16_t	number
proton.uint	uint32_t	number
proton.ulong	uint64_t	number

AMQ Python type before encoding	AMQ C++ type	AMQ JavaScript type
<code>proton.float32</code>	<code>float</code>	<code>number</code>
<code>float</code>	<code>double</code>	<code>number</code>
<code>proton.Array</code>	-	<code>Array</code>
<code>list</code>	<code>std::vector</code>	<code>Array</code>
<code>dict</code>	<code>std::map</code>	<code>object</code>
<code>uuid.UUID</code>	<code>proton::uuid</code>	<code>number</code>
<code>proton.symbol</code>	<code>proton::symbol</code>	<code>string</code>
<code>proton.timestamp</code>	<code>proton::timestamp</code>	<code>number</code>

Table 7.4. AMQ Python and other AMQ client types (2 of 2)

AMQ Python type before encoding	AMQ .NET type	AMQ Ruby type
<code>None</code>	<code>null</code>	<code>nil</code>
<code>bool</code>	<code>System.Boolean</code>	<code>true, false</code>
<code>proton.char</code>	<code>System.Char</code>	<code>String</code>
<code>unicode</code>	<code>System.String</code>	<code>String</code>
<code>bytes</code>	<code>System.Byte[]</code>	<code>String</code>
<code>proton.byte</code>	<code>System.SByte</code>	<code>Integer</code>
<code>proton.short</code>	<code>System.Int16</code>	<code>Integer</code>
<code>proton.int32</code>	<code>System.Int32</code>	<code>Integer</code>
<code>long</code>	<code>System.Int64</code>	<code>Integer</code>
<code>proton.ubyte</code>	<code>System.Byte</code>	<code>Integer</code>
<code>proton.ushort</code>	<code>System.UInt16</code>	<code>Integer</code>
<code>proton.uint</code>	<code>System.UInt32</code>	<code>Integer</code>

AMQ Python type before encoding	AMQ .NET type	AMQ Ruby type
<code>proton.ulong</code>	<code>System.UInt64</code>	<code>Integer</code>
<code>proton.float32</code>	<code>System.Single</code>	<code>Float</code>
<code>float</code>	<code>System.Double</code>	<code>Float</code>
<code>proton.Array</code>	-	<code>Array</code>
<code>list</code>	<code>Amqp.List</code>	<code>Array</code>
<code>dict</code>	<code>Amqp.Map</code>	<code>Hash</code>
<code>uuid.UUID</code>	<code>System.Guid</code>	-
<code>proton.symbol</code>	<code>Amqp.Symbol</code>	<code>Symbol</code>
<code>proton.timestamp</code>	<code>System.DateTime</code>	<code>Time</code>

7.2. 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 Python 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 7.5. AMQ Python and JMS message types

AMQ Python body type	JMS message type
<code>unicode</code>	<code>TextMessage</code>
<code>None</code>	<code>TextMessage</code>
<code>bytes</code>	<code>BytesMessage</code>
Any other type	<code>ObjectMessage</code>

7.3. 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](#).

7.4. 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.

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 2020-02-26 17:20:12 UTC