Red Hat AMQ Clients 2.11

Using the AMQ Python Client

For Use with AMQ Clients 2.11

Last Updated: 2022-04-26

Red Hat AMQ Clients 2.11 Using the AMQ Python Client

For Use with AMQ Clients 2.11

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

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

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 2.11 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
- Seamless conversion between AMQP and language-native data types
- Access to all the features and capabilities of AMQP 1.0
- 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/.

1.2. SUPPORTED STANDARDS AND PROTOCOLS

AMQ Python 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

Refer to Red Hat AMQ Supported Configurations on the Red Hat Customer Portal for current information regarding AMQ Python 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. 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.

1.5. 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:

```
$ cd <project-dir>
```

CHAPTER 2. INSTALLATION

This chapter guides you through the steps to install AMQ Python 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 Python, you must install Python 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. Replace ***<version>*** with **2** for the main release stream or **2.11** for the long term support release stream. 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-<version>-for-rhel-7-<variant>-rpms
   ```

   Red Hat Enterprise Linux 8

   ```
   $ sudo subscription-manager repos --enable=amq-clients-<version>-for-rhel-8-x86_64-rpms
   ```

2. Use the **yum** command to install the packages.

   Main release stream

   ```
   $ sudo yum install python3-qpid-proton python-qpid-proton-docs
   ```

   AMQ Clients 2.11 long term support stream

   ```
   $ sudo yum install python-qpid-proton python-qpid-proton-docs
   ```

For more information about using packages, see Appendix B, *Using Red Hat Enterprise Linux packages* .

2.3. INSTALLING ON MICROSOFT WINDOWS

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 Clients** entry in the **INTEGRATION AND AUTOMATION** category.

3. Click **Red Hat AMQ Clients**. The **Software Downloads** page opens.

4. Download the **AMQ Clients 2.11.0 Python** .whl file for your Python version.

Python 3.6	python_qpid_proton-0.37.0-cp36-cp36m-win_amd64.whl
Python 3.8	python_qpid_proton-0.37.0-cp38-cp38-win_amd64.whl

5. Open a command prompt window and use the **pip install** command to install the .whl file.

Python 3.6

```
> pip install python_qpid_proton-0.37.0-cp36-cp36m-win_amd64.whl
```

Python 3.8

```
> pip install python_qpid_proton-0.37.0-cp38-cp38-win_amd64.whl
```

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.py** example.

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

3.3. RUNNING HELLO WORLD ON MICROSOFT WINDOWS

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.

Download and run the Hello World example.

```
> curl -o helloworld.py https://raw.githubusercontent.com/apache/qpid-
proton/master/python/examples/helloworld.py
> 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 and the Qpid Proton Python examples.

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. For more information, see Chapter 3, *Getting started*.

```
$ 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. For more information, see Chapter 3, *Getting started*.

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

CHAPTER 5. USING THE API

For more information, see the AMQ Python API reference and AMQ Python example suite .

5.1. 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.

5.2. ACCESSING EVENT-RELATED OBJECTS

The **event** argument has attributes for accessing the object the event is regarding. For example, the **on_connection_opened** event sets the event **connection** attribute.

In addition to the primary object for the event, all objects that form the context for the event are set as well. Attributes with no relevance to a particular event are null.

Example: Accessing event-related objects

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

5.3. 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()
```

5.4. 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 constucted.

CHAPTER 6. NETWORK CONNECTIONS

6.1. 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
```

6.2. 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")
```

For information about creating secure connections, see Chapter 7, *Security*.

6.3. 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.

6.4. 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.

6.5. ACCEPTING INCOMING CONNECTIONS

AMQ Python can accept inbound network connections, enabling you to build custom 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: Accepting 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.

CHAPTER 7. SECURITY

7.1. 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")
```

7.2. 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")
```

7.3. 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)
```

7.4. 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.

CHAPTER 8. 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.

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

8.1. 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

8.2. 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.

8.3. 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.

CHAPTER 9. MESSAGE DELIVERY

9.1. 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.

9.2. 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")
```

9.3. 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.

9.4. 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 explicity acknowledge a delivery, then the library accepts it after **on_message** returns. To disable this behavior, set the **auto_accept** receiver option to false.

CHAPTER 10. ERROR HANDLING

Errors in AMQ Python can be handled in two different ways:

- Catching exceptions
- Overriding event-handling functions to intercept AMQP protocol or connection errors

10.1. CATCHING EXCEPTIONS

All of the exceptions that AMQ Python throws inherit from the **ProtonException** class, which in turn inherits from the Python **Exception** class.

The following example illustrates how to catch any exception thrown from AMQ Python:

Example: API-specific exception handling

```
try:
    # Something that might throw an exception
except ProtonException as e:
    # Handle Proton-specific problems here
except Exception as e:
    # Handle more general problems here
}
```

If you do not require API-specific exception handling, you only need to catch **Exception**, since **ProtonException** inherits from it.

10.2. HANDLING CONNECTION AND PROTOCOL ERRORS

You can handle protocol-level errors by overriding the following **messaging_handler** methods:

- **on_transport_error(event)**
- **on_connection_error(event)**
- **on_session_error(event)**
- **on_link_error(event)**

These event-handling functions are called whenever there is an error condition with the specific object that is in the event. After calling the error handler, the appropriate close handler is also called.

NOTE

Because the close handlers are called in the event of any error, only the error itself needs to be handled within the error handler. Resource cleanup can be managed by close handlers. If there is no error handling that is specific to a particular object, it is typical to use the general **on_error** handler and not have a more specific handler.

NOTE

When reconnect is enabled and the remote server closes a connection with the **amqp:connection:forced** condition, the client does not treat it as an error and thus does not fire the **on_connection_error** handler. The client instead begins the reconnection process.

CHAPTER 11. LOGGING

11.1. 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.

CHAPTER 12. DISTRIBUTED TRACING

12.1. ENABLING DISTRIBUTED 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 13. 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.

13.1. 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.

13.2. THE 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"
    }
}
```

13.3. CONFIGURATION OPTIONS

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

Table 13.1. 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	*None*	The user name for authentication
password	string	*None*	The password for authentication
sasl.mechanisms	list or string	*None (system defaults)*	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	*None*	The filename or database ID of the client certificate
tls.key	string	*None*	The filename or database ID of the private key for the client certificate
tls.ca	string	*None*	The filename, directory, or database ID of the CA certificate
tls.verify	boolean	**true**	Require a valid server certificate with a matching hostname

CHAPTER 14. 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.

14.1. 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 14.1. 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 14.2. 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 14.3. 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
proton.float32	**float**	**number**
float	**double**	**number**
proton.Array	-	**Array**
list	**std::vector**	**Array**
dict	**std::map**	**object**
uuid.UUID	**proton::uuid**	**number**
proton.symbol	**proton::symbol**	**string**
proton.timestamp	**proton::timestamp**	**number**

Table 14.4. AMQ Python and other AMQ client types (2 of 2)

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

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

14.2. 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 14.5. AMQ Python and JMS message types

AMQ Python body type	JMS message type
unicode	**TextMessage**
None	**TextMessage**
bytes	**BytesMessage**
Any other type	**ObjectMessage**

14.3. 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.

14.4. 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 for this product 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 about registering your system, see one of the following resources:

- Red Hat Enterprise Linux 7 - Registering the system and managing subscriptions
- Red Hat Enterprise Linux 8 - Registering the system and managing subscriptions

APPENDIX B. USING RED HAT ENTERPRISE LINUX PACKAGES

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

To ensure the RPM packages for this product are available, you must first register your system.

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 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 Python 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

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

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

Revised on 2022-04-26 14:27:55 UTC