



Red Hat AMQ 2021.Q3

AMQ Python クライアントの使用

AMQ Clients 2.10 向け

Red Hat AMQ 2021.Q3 AMQ Python クライアントの使用

AMQ Clients 2.10 向け

Enter your first name here. Enter your surname here.

Enter your organisation's name here. Enter your organisational division here.

Enter your email address here.

法律上の通知

Copyright © 2021 | You need to change the HOLDER entity in the en-US/Using_the_AMQ_Python_Client.ent file |.

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.

概要

本ガイドでは、クライアントのインストールや設定、実例の実行、他の AMQ コンポーネントでのクライアントの使用方法について説明します。

目次

多様性を受け入れるオープンソースの強化	4
第1章 概要	5
1.1. 主な特長	5
1.2. サポート対象の標準およびプロトコル	5
1.3. サポートされる構成	6
1.4. 用語および概念	6
1.5. 本書の表記慣例	7
sudo コマンド	7
ファイルパス	7
変数テキスト	7
第2章 インストール	8
2.1. 前提条件	8
2.2. 「INSTALLING ON RED HAT ENTERPRISE LINUX」を参照してください。	8
2.3. 「INSTALLING ON MICROSOFT WINDOWS」を参照してください。	8
第3章 スタートガイド	10
3.1. 前提条件	10
3.2. RED HAT ENTERPRISE LINUX での HELLO WORLD の実行	10
3.3. MICROSOFT WINDOWS での HELLO WORLD の実行	10
第4章 例	11
4.1. メッセージの送信	11
サンプルの実行	12
4.2. メッセージの受信	12
サンプルの実行	13
第5章 API の使用	14
5.1. メッセージングイベントの処理	14
5.2. イベント関連のオブジェクトへのアクセス	14
5.3. コンテナの作成	14
5.4. コンテナアイデンティティの設定	15
第6章 ネットワーク接続	16
6.1. 接続 URL	16
6.2. 外向き接続の作成	16
6.3. 再接続の設定	16
6.4. フェイルオーバーの設定	17
6.5. 受信接続の許可	17
第7章 セキュリティー	19
7.1. SSL/TLS を使用した接続のセキュリティー保護	19
7.2. ユーザーとパスワードを使用した接続	19
7.3. SASL 認証の設定	19
7.4. KERBEROS を使用した認証	20
第8章 送信者およびレシーバー	21
8.1. オンデマンドでキューとトピックの作成	21
8.2. 永続サブスクリプションの作成	22
8.3. 共有サブスクリプションの作成	22
第9章 メッセージ配信	24
9.1. メッセージの送信	24

9.2. 送信されたメッセージの追跡	24
9.3. メッセージの受信	24
9.4. 受信したメッセージの承認	25
第10章 エラー処理	26
10.1. 例外のキャッチ	26
10.2. 接続およびプロトコルエラーの処理	26
第11章 ロギング	28
11.1. プロトコルロギングの有効化	28
第12章 分散トレーシング	29
12.1. 分散トレースの有効化	29
第13章 ファイルベースの設定	30
13.1. ファイルの場所	30
13.2. ファイルフォーマット	30
13.3. SSL 設定オプション	31
第14章 相互運用性	33
14.1. 他の AMQP クライアントとの相互運用	33
14.2. AMQ JMS での相互運用	37
JMS メッセージタイプ	37
14.3. AMQ BROKER への接続	37
14.4. AMQ INTERCONNECT への接続	38
付録A サブスクリプションの使用	39
A.1. アカウントへのアクセス	39
A.2. サブスクリプションのアクティベート	39
A.3. リリースファイルのダウンロード	39
A.4. パッケージを受信するためのシステムの登録	39
付録B RED HAT ENTERPRISE LINUX パッケージの使用	41
B.1. 概要	41
B.2. パッケージの検索	41
B.3. パッケージのインストール	41
B.4. パッケージ情報のクエリー	41
付録C サンプルでの AMQ BROKER の使用	43
C.1. ブローカーのインストール	43
C.2. ブローカーの起動	43
C.3. キューの作成	43
C.4. ブローカーの停止	44

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。これは大規模な取り組みであるため、これらの変更は今後の複数のリリースで段階的に実施されます。詳細は、[Red Hat CTO である Chris Wright のメッセージ](#)をご覧ください。

第1章 概要

AMQ Python は、メッセージングアプリケーションを開発するためのライブラリーです。AMQP メッセージを送受信する Python アプリケーションを作成できます。

AMQ Python は、複数の言語やプラットフォームをサポートするメッセージングライブラリースイートである AMQ Clients の一部です。クライアントの概要は、「[AMQ Clients Overview](#)」を参照してください。本リリースに関する詳細は、『[AMQ Clients 2.10 Release Notes](#)』を参照してください。

AMQ Python は、[Apache Qpid](#) の Proton API をベースとしています。詳細な API ドキュメントは、「[AMQ Python API リファレンス](#)」を参照してください。

1.1. 主な特長

- 既存のアプリケーションとの統合を簡素化するイベント駆動型の API
- セキュアな通信用の SSL/TLS
- 柔軟な SASL 認証
- 自動再接続およびフェイルオーバー
- AMQP と言語ネイティブデータ型間のシームレスな変換
- AMQP 1.0 のすべての機能と機能へのアクセス
- OpenTracing 標準(RHEL 7 および 8) に基づいた分散トレーシング機能



重要

AMQ Clients での分散トレーシングはテクノロジープレビュー機能です。テクノロジープレビューの機能は、Red Hat の実稼働環境のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat では、これらについて実稼働環境での使用を推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストやフィードバックの提供を可能にするために提供されます。Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、<https://access.redhat.com/ja/support/offerings/techpreview> を参照してください。

1.2. サポート対象の標準およびプロトコル

AMQ Python は、以下の業界標準およびネットワークプロトコルをサポートします。

- [Advanced Message Queueing Protocol \(AMQP\)](#) のバージョン 1.0
- SSL の後継である TLS ([Transport Layer Security](#)) プロトコルのバージョン 1.0、1.1、1.2、および 1.3
- ANONYMOUS、PLAIN、SCRAM、EXTERNAL、および GSSAPI (Kerberos) を含む、[Cyrus SASL](#) でサポートされる [単純な認証およびセキュリティーレイヤー \(SASL\) メカニズム](#)
- IPv6 での最新の TCP

1.3. サポートされる構成

AMQ Python でサポートされている設定については、Red Hat カスタマーポータルの「[Red Hat AMQ 7 でサポートされる構成](#)」を参照してください。

1.4. 用語および概念

本セクションでは、コア API エンティティを紹介し、それらが一緒に操作する方法を説明します。

表1.1 API の用語

エンティティ	説明
コンテナ	接続の最上位のコンテナ。
接続	ネットワーク上の2つのピア間の通信用のチャンネル。これにはセッションが含まれます。
セッション	メッセージの送受信を行うためのコンテキスト。送信者およびレシーバーが含まれます。
送信	メッセージをターゲットに送信するためのチャンネル。これにはターゲットがあります。
受信	ソースからメッセージを受信するためのチャンネル。ソースがあります。
ソース	メッセージに対する名前付きポイント。
ターゲット	メッセージの名前付き宛先。
メッセージ	アプリケーション固有の情報部分。
配信	メッセージの転送

AMQ Python は **メッセージ** を送受信します。メッセージは、**送信側** と **受信側** を介して接続されたピア間で転送されます。送信側およびレシーバーは **セッション** 上で確立されます。セッションは **コネクション** を介して確立されます。接続は、一意に識別された2つの **コンテナ** 間で確立されます。コネクションには複数のセッションを含めることができますが、多くの場合、これは必要ありません。API を使用すると、セッションが必要でない限り、セッションを無視できます。

送信ピアは、メッセージを送信するために送信者を作成します。送信側には、リモートピアでキューまたはトピックを識別する **ターゲット** があります。受信ピアは、メッセージを受信するための受信側を作成します。受信側には、リモートピアでキューまたはトピックを識別する **ソース** があります。

メッセージの送信は、**配信** と呼ばれます。メッセージは送信される内容で、ヘッダーやアノテーションなどのすべてのメタデータが含まれます。配信は、そのコンテンツの移動に関連するプロトコルエクステンションです。

配信が完了したことを示すには、送信側または受信側セットのいずれかです。これが設定されていることを知らせると、その配信に関する通信はなくなります。受信側は、メッセージを受諾または拒否するかどうかを指定することもできます。

1.5. 本書の表記慣例

sudo コマンド

本書では、root 権限を必要とするコマンドには **sudo** が使用されています。何らかの変更がシステム全体に影響する可能性があるため、**sudo** を使用する場合は注意が必要です。**sudo** の詳細は、「[sudo コマンドの使用](#)」を参照してください。

ファイルパス

本書では、すべてのファイルパスが Linux、UNIX、および同様のオペレーティングシステムで有効です (例: `/home/andrea`)。Microsoft Windows では、同等の Windows パスを使用する必要があります (例: `C:\Users\andrea`)。

変数テキスト

本書には、実際の環境に固有の値に置き換える必要がある変数を含むコードブロックが含まれています。変数テキストは中括弧で囲まれ、斜体の等幅フォントとしてスタイル設定されます。たとえば、以下の例では、`<project-dir>` を実際の環境の値に置き換えます。

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

第2章 インストール

本章では、環境に AMQ Python をインストールする手順を説明します。

2.1. 前提条件

- AMQ リリースファイルおよびリポジトリにアクセスするには、[サブスクリプション](#) が必要です。
- Red Hat Enterprise Linux にパッケージをインストールするには、[システムを登録する](#) 必要があります。
- AMQ Python を使用するには、お使いの環境に Python をインストールする必要があります。

2.2. 「INSTALLING ON RED HAT ENTERPRISE LINUX」を参照してください。

手順

1. **subscription-manager** コマンドを使用して、必要なパッケージリポジトリをサブスクライブします。メジャーリリースストリームの **<version>** を **2**、または長期サポートのリリースストリームの場合は **2.9** に置き換えます。必要に応じて、**<variant>** を Red Hat Enterprise Linux のバリエーションの値 (例: **server** または **workstation**) に置き換えます。

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. **yum** コマンドを使用してパッケージをインストールします。

メインリリースストリーム

```
$ sudo yum install python3-qpiproton python-qpiproton-docs
```

AMQ Clients 2.9 の長期サポートストリーム

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

パッケージの使用方法は、[付録B Red Hat Enterprise Linux パッケージの使用](#) を参照してください。

2.3. 「INSTALLING ON MICROSOFT WINDOWS」を参照してください。

手順

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。

2. INTEGRATION AND AUTOMATION カテゴリで **Red Hat AMQ Client** エントリーを見つけます。
3. **Red Hat AMQ Clients** をクリックします。 **Software Downloads** ページが開きます。
4. お使いの Python バージョン用の **AMQ Clients 2.10.0 Python**.whl ファイルをダウンロードします。

Python 3.6	python_qpid_proton-0.35.0-cp36-cp36m-win_amd64.whl
Python 3.8	python_qpid_proton-0.35.0-cp38-cp38-win_amd64.whl

5. コマンドプロンプトを開き、 **pip install** コマンドを使用して .whl ファイルをインストールします。

Python 3.6

```
> pip install python_qpid_proton-0.35.0-cp36-cp36m-win_amd64.whl
```

Python 3.8

```
> pip install python_qpid_proton-0.35.0-cp38-cp38-win_amd64.whl
```

第3章 スタートガイド

本章では、環境を設定して簡単なメッセージングプログラムを実行する手順を説明します。

3.1. 前提条件

- お使いの環境の [インストール](#) 手順を完了する必要があります。
- インターフェース **localhost** およびポート **5672** で接続をリッスンする AMQP 1.0 メッセージブローカーが必要です。匿名アクセスを有効にする必要があります。詳細は、「[ブローカーの開始](#)」を参照してください。
- **examples** という名前のキューが必要です。詳細は、「[キューの作成](#)」を参照してください。

3.2. RED HAT ENTERPRISE LINUX での HELLO WORLD の実行

Hello World の例では、ブローカーへの接続を作成し、グリーティングが含まれるメッセージを **examples** キューに送信し、それを受け取ります。成功すると、受け取ったメッセージをコンソールに出力します。

`examples` ディレクトリーに移動し、**helloworld.py** の例を実行します。

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

3.3. MICROSOFT WINDOWS での HELLO WORLD の実行

Hello World の例では、ブローカーへの接続を作成し、グリーティングが含まれるメッセージを **examples** キューに送信し、それを受け取ります。成功すると、受け取ったメッセージをコンソールに出力します。

Hello World の例をダウンロードして実行します。

```
> curl -o helloworld.py https://raw.githubusercontent.com/apache/qpid-proton/master/python/examples/helloworld.py  
> python helloworld.py  
Hello World!
```

第4章 例

本章では、サンプルプログラムで AMQ Python を使用方法について説明します。

その他の例は、[AMQ Python サンプルのスイート](#) と [Qpid Proton Python の例](#) を参照してください。

4.1. メッセージの送信

このクライアントプログラムは、<connection-url> を使用してサーバーに接続します。ターゲット <address> の送信側は <message-body> が含まれるメッセージを送信し、接続を閉じて終了します。

例: メッセージの送信

```

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

```

サンプルの実行

サンプルプログラムを実行するには、これをローカルファイルにコピーし、**python** コマンドを使用してこれを呼び出します。詳細は、[3章スタートガイド](#)を参照してください。

```
$ python send.py amqp://localhost queue1 hello
```

4.2. メッセージの受信

このクライアントプログラムは **<connection-url>** を使用してサーバーに接続し、ソース **<address>** のレシーバーを作成し、終了するか **<count>** メッセージに到達するまでメッセージを受信します。

例: メッセージの受信

```

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
```

サンプルの実行

サンプルプログラムを実行するには、これをローカルファイルにコピーし、**python** コマンドを使用してこれを呼び出します。詳細は、[3章 スタートガイド](#)を参照してください。

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

第5章 API の使用

詳細は、「[AMQ Python API リファレンス](#)」および「[AMQ Python サンプルスイート](#)」を参照してください。

5.1. メッセージングイベントの処理

AMQ Python は非同期イベント駆動型 API です。アプリケーションがイベントを処理する方法を定義するために、ユーザーは **MessagingHandler** クラスでコールバックメソッドを実装します。これらのメソッドは、ネットワークアクティビティとして呼び出され、タイマーが新規イベントをトリガーします。

例: メッセージングイベントの処理

```
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")
```

これらはいくつかの一般的なケースイベントのみです。完全セットは [API リファレンス](#) に文書化されています。

5.2. イベント関連のオブジェクトへのアクセス

event 引数には、イベントが関連するオブジェクトにアクセスするための属性があります。たとえば、**on_connection_opened** イベントはイベント **connection** 属性を設定します。

イベントのプライマリーオブジェクトに加えて、イベントのコンテキストを形成するすべてのオブジェクトも設定されます。特定のイベントに対する関連性のない属性は null です。

例: イベント関連のオブジェクトへのアクセス

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

5.3. コンテナの作成

コンテナはトップレベルの API オブジェクトです。これは、接続を作成するエントリーポイントであり、メインのイベントループを実行します。多くの場合、これはグローバルイベントハンドラーで構築されます。

例: コンテナの作成

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

5.4. コンテナアイデンティティの設定

各コンテナインスタンスには、コンテナ ID と呼ばれる一意のアイデンティティがあります。AMQ Python が接続を確立すると、コンテナ ID をリモートピアに送信します。コンテナ ID を設定するには、これを **Container** コンストラクターに渡します。

例: コンテナアイデンティティの設定

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

ユーザーが ID を設定しないと、コンテナが構成されると、ライブラリーは UUID を生成します。

第6章 ネットワーク接続

6.1. 接続 URL

connection URL は、新規接続の確立に使用される情報をエンコードします。

接続 URL 構文

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

- **スキーム**: 暗号化されていない TCP の **amqp**、または SSL/TLS 暗号化による TCP の **amqps** のいずれかの接続トランスポート。
- **ホスト**: リモートのネットワークホスト。値には、ホスト名または数値の IP アドレスを指定できます。IPv6 アドレスは角括弧で囲む必要があります。
- **port**: リモートネットワークポート。この値はオプションです。デフォルト値は、**amqp** スキームの場合は 5672 で、**amqps** スキームの場合は 5671 です。

接続 URL の例

```
amqps://example.com  
amqps://example.net:56720  
amqp://127.0.0.1  
amqp://[::1]:2000
```

6.2. 外向き接続の作成

リモートサーバーに接続するには、[接続 URL](#) を使用して **Container.connect()** メソッドを呼び出します。通常、これは **MessagingHandler.on_start()** メソッド内で行われます。

例: 送信接続の作成

```
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")
```

セキュアな接続の作成に関する詳細は、[7章 セキュリティー](#) を参照してください。

6.3. 再接続の設定

再接続すると、クライアントが失われた接続から回復できます。これは、一時的なネットワークまたはコンポーネントの障害後に、分散システムのコンポーネントが再確立された状態にするために使用されます。

AMQ Python はデフォルトで再接続を有効にします。接続が失われたり、接続試行に失敗した場合に、クライアントは簡単な遅延後に再度試行します。遅延は、デフォルトの最大値 10 秒まで、新しい試行ごとに指数関数的に増加します。

再接続を無効にするには、**reconnect** 接続オプションを **False** に設定します。

例: 再接続の無効化

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

接続試行間の遅延を制御するには、**reset()** メソッドおよび **next()** メソッドを実装するクラスを定義し、**reconnect** 接続オプションをそのクラスのインスタンスに設定します。

例: 再接続の設定

```
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())
```

next メソッドは、次の遅延を秒単位で返します。**reset** メソッドは、再接続プロセスが開始する前に一度呼び出されます。

6.4. フェイルオーバーの設定

AMQ Python では、複数の接続エンドポイントを設定できます。ある接続に失敗すると、クライアントはリスト内の次の接続を試みます。一覧に使い切られると、プロセスは最初から開始します。

複数の接続エンドポイントを指定するには、**urls** 接続オプションを接続 URL の一覧に設定します。

例: フェイルオーバーの設定

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

url オプションおよび **urls** オプションを同時に使用するのはエラーです。

6.5. 受信接続の許可

AMQ Python はインバウンドネットワーク接続を受け入れ、カスタムメッセージングサーバーを構築できます。

接続のリッスンを開始するには、ローカルホストアドレスおよびリッスンするポートが含まれる URL で **Container.listen()** メソッドを使用します。

例: 受信接続の許可

```
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)
```

特別な IP アドレス **0.0.0.0** は、利用可能なすべての IPv4 インターフェイスでリッスンします。すべての IPv6 インターフェイスをリッスンするには、**:::0** を使用します。

詳細は、[サーバー receive.py の例](#) を参照してください。

第7章 セキュリティー

7.1. SSL/TLS を使用した接続のセキュリティー保護

AMQ Python は SSL/TLS を使用して、クライアントとサーバー間の通信を暗号化します。

SSL/TLS を使用してリモートサーバーに接続するには、**amqps** スキームで接続 URL を使用します。

例: SSL/TLS の有効化

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

7.2. ユーザーとパスワードを使用した接続

AMQ Python は、ユーザーとパスワードによる接続を認証できます。

認証に使用する認証情報を指定するには、**connect()** メソッドで **user** および **password** オプションを設定します。

例: ユーザーとパスワードを使用した接続

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

7.3. SASL 認証の設定

AMQ Python は SASL プロトコルを使用して認証を実行します。SASL は多くの異なる認証メカニズムを使用できます。2つのネットワークピアに接続すると、許可されるメカニズムを交換し、両方で許可される最も強力なメカニズムが選択されます。

注記

クライアントは Cyrus SASL を使用して認証を実行します。Cyrus SASL は、プラグインを使用して特定の SASL メカニズムをサポートします。特定の SASL メカニズムを使用する前に、関連するプラグインをインストールする必要があります。たとえば、SASL PLAIN 認証を使用するには、**cyrus-sasl-plain** プラグインが必要です。

Red Hat Enterprise Linux の Cyrus SASL プラグインのリストを表示するには、**yum search cyrus-sasl** コマンドを使用します。Cyrus SASL プラグインをインストールするには、**yum install PLUG-IN** コマンドを使用します。

デフォルトでは、AMQ Python はローカル SASL ライブラリー設定でサポートされるすべてのメカニズムを許可します。許可されるメカニズムを制限し、ネゴシエートできるメカニズムを制御するには、**allowed_mechs** コネクションオプションを使用します。スペースで区切ったメカニズム名のリストが含まれる文字列を受け入れます。

例: SASL 認証の設定

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

この例では、サーバーが他のオプションを提供するように接続しても、**ANONYMOUS** メカニズムを使用した認証を強制します。有効なメカニズムには、**ANONYMOUS**、**PLAIN**、**SCRAM-SHA-256**、**SCRAM-SHA-1**、**GSSAPI**、および **EXTERNAL** が含まれます。

AMQ Python はデフォルトで SASL を有効にします。これを無効にするには、**sasl_enabled** 接続オプションを `false` に設定します。

例: SASL の無効化

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

7.4. KERBEROS を使用した認証

Kerberos は、暗号化されたチケットの交換に基づいて一元管理された認証用のネットワークプロトコルです。詳細は、「[Kerberos の使用](#)」を参照してください。

1. お使いのオペレーティングシステムで Kerberos を設定します。「[Red Hat Enterprise Linux で Kerberos を設定する](#)」を参照してください。
2. クライアントアプリケーションで **GSSAPI** SASL メカニズムを有効にします。

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

3. **kinit** コマンドを使用して、ユーザーの認証情報を認証し、作成された Kerberos チケットを保存します。

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

4. クライアントプログラムを実行します。

第8章 送信者およびレシーバー

クライアントは送信側と受信側のリンクを使用して、メッセージの配信にチャンネルを表します。送信者と受信側は一方方向で、メッセージの送信元はソースエンド、メッセージの送信先はターゲットエンドとなります。

ソースとターゲットは、多くの場合、メッセージブローカーのキューまたはトピックを参照します。ソースは、サブスクリプションを表すためにも使用されます。

8.1. オンデマンドでキューとトピックの作成

一部のメッセージサーバーは、キューとトピックのオンデマンド作成をサポートします。送信側またはレシーバーが割り当てられている場合、サーバーは送信側のターゲットアドレスまたは受信側ソースアドレスを使用して、アドレスに一致する名前を持つキューまたはトピックを作成します。

メッセージサーバーは通常、キュー（1対1のメッセージ配信用）またはトピック（1対多のメッセージ配信の場合）を作成します。クライアントは、ソースまたはターゲットに **queue** または **topic** 機能を設定することで、希望のものを指定できます。

キューまたはトピックセマンティクスを選択するには、以下の手順に従います。

1. キューとトピックの自動作成のために、メッセージサーバーを設定します。多くの場合、これはデフォルト設定になります。
2. 以下の例のように、送信側のターゲットまたは受信側ソースに **queue** または **topic** 機能を設定します。

例：オンデマンドで作成されたキューへの送信

```
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())
```

例: 要求時に作成されたトピックからの受信

```
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())
```

詳細は、以下の例を参照してください。

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

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

8.2. 永続サブスクリプションの作成

永続サブスクリプションは、メッセージの受信側を表すリモートサーバーの状態です。通常、クライアントが閉じられると、メッセージ受信側は破棄されます。ただし、永続サブスクリプションは永続的であるため、クライアントはこれらのサブスクリプションの割り当てを解除してから、後で再度アタッチすることができます。デタッチ中に受信したすべてのメッセージは、クライアントの再割り当て時に利用できます。

永続サブスクリプションは、クライアントコンテナ ID とレシーバー名を組み合わせることでサブスクリプション ID を形成することで一意に識別されます。サブスクリプションが回復できるようにするには、これらの値に安定した値が必要です。

永続サブスクリプションを作成するには、以下の手順に従います。

1. 接続コンテナ ID を **client-1** などの安定した値に設定します。

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

2. **durability** プロパティおよび **expiry_policy** プロパティを設定して、受信側のソースを耐久性のあるものにします。

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

3. **sub-1** などの安定した名前で作成し、ソースプロパティを適用します。

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

サブスクリプションからデタッチするには、**Receiver.detach()** メソッドを使用します。サブスクリプションを終了するには、**Receiver.close()** メソッドを使用します。

詳細は、[durable-subscribe.py](#) の例を参照してください。

8.3. 共有サブスクリプションの作成

共有サブスクリプションとは、1つ以上のメッセージレシーバーを表すリモートサーバーの状態のことです。共有されているので、複数のクライアントは同じメッセージのストリームから消費できます。

クライアントは、レシーバーソースに **shared** 機能を設定して、共有サブスクリプションを設定します。

共有サブスクリプションは、クライアントコンテナ ID とレシーバー名を組み合わせることでサブスクリプション ID を形成することで一意に識別されます。複数のクライアントプロセスで同じサブスクリプションを見つけることができるように、これらの値が安定している必要があります。**shared** に加えて **global** 機能が設定されている場合、レシーバー名のみを使用してサブスクリプションを特定します。

永続サブスクリプションを作成するには、以下の手順に従います。

1. 接続コンテナ ID を **client-1** などの安定した値に設定します。

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

2. **shared** 機能を設定して、共有用の受信側ソースを設定します。

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

3. **sub-1** などの安定した名前で作成し、ソースプロパティを適用します。

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

サブスクリプションからデタッチするには、**Receiver.detach()** メソッドを使用します。サブスクリプションを終了するには、**Receiver.close()** メソッドを使用します。

詳細は、[shared-subscribe.py](#) の例を参照してください。

第9章 メッセージ配信

9.1. メッセージの送信

メッセージを送信するには、`on_sendable` イベントハンドラーを上書きし、`Sender.send()` メソッドを呼び出します。`sendable` イベントは、`Sender` が少なくとも1つのメッセージを送信するのに十分なクレジットがあると実行されます。

例: メッセージの送信

```
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)
```

詳細は、[send.py](#) の例を参照してください。

9.2. 送信されたメッセージの追跡

メッセージを送信すると、送信側は転送を表す `delivery` オブジェクトへの参照を維持することができます。メッセージが配信されると、受信側はこれを受け入れるか、拒否します。各配信の結果が送信者に通知されます。

送信されたメッセージの結果を監視するには、`on_accepted` および `on_rejected` イベントハンドラーを上書きし、配信状態の更新を `send()` から返された配信にマップします。

例: 送信したメッセージの追跡

```
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. メッセージの受信

メッセージの受信には、レシーバーを作成し、`on_message` イベントハンドラーを上書きします。

例: メッセージの受信

```
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)
```

詳細は、[receive.py](#) の例を参照してください。

9.4. 受信したメッセージの承認

配信を明示的に許可または拒否するには、**on_message** イベントハンドラーの **ACCEPTED** 状態または **REJECTED** 状態の **Delivery.update()** メソッドを使用します。

例: 受信したメッセージの承認

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

デフォルトでは、配信が明示的な承認されない場合、ライブラリーは **on_message** の戻り値後に受け入れます。この動作を無効にするには、**auto_accept** レシーバーオプションを **false** に設定します。

第10章 エラー処理

AMQ Python でのエラーは、以下の 2 つの方法で処理できます。

- 例外のキャッチ
- AMQP プロトコルまたは接続エラーを傍受するためのイベント処理の関数の上書き

10.1. 例外のキャッチ

AMQ Python が出力するすべての例外は **ProtonException** クラスを継承しており、そのクラスは Python の **Exception** クラスを継承しています。

以下の例は、AMQ Python から発生した例外をキャッチする方法を示しています。

例: API 固有の例外処理

```
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
}
```

API 固有の例外処理が必要ない場合は、**ProtonException** が継承されてから **Exception** のみをキャッチする必要があります。

10.2. 接続およびプロトコルエラーの処理

以下の **messaging_handler** メソッドを上書きすると、プロトコルレベルのエラーを処理することができます。

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

これらのイベント処理機能は、イベントにある特定のオブジェクトとともにエラー状態が生じるたびに呼び出されます。エラーハンドラーを呼び出すと、適切なクローズハンドラーも呼び出されます。



注記

クローズハンドラーはエラー発生時に呼び出されるため、エラーハンドラー内でのみ処理する必要があります。リソースクリーンアップは、クローズハンドラーで管理できます。特定のオブジェクトに固有のエラー処理がない場合、一般的な **on_error** ハンドラーを使用することが一般的で、より具体的なハンドラーはありません。



注記

再接続が有効になり、リモートサーバーが **amqp:connection:forced** 条件で接続を閉じると、クライアントはこれをエラーとして処理しないため、**on_connection_error** ハンドラーは実行されません。代わりに、クライアントが再接続プロセスを開始します。

第11章 ロギング

11.1. プロトコルロギングの有効化

クライアントは AMQP プロトコルフレームをコンソールに記録できます。通常、このデータは問題を診断する際に重要です。

プロトコルロギングを有効にするには、**PN_TRACE_FRM** 環境変数を **1** に設定します。

例: プロトコルロギングの有効化

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

プロトコルロギングを無効にするには、**PN_TRACE_FRM** 環境変数の設定を解除します。

第12章 分散トレーシング

12.1. 分散トレースの有効化

クライアントは、OpenTracing 標準の Jaeger 実装に基づいて分散トレーシングを提供します。アプリケーションでトレースを有効にするには、以下の手順に従います。

1. トレース依存関係をインストールします。

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. プログラムにグローバルトレーサーを登録します。

例: グローバルトレーサー設定

```
from proton.tracing import init_tracer

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

Jaeger の設定に関する詳細は、「[Jaeger Sampling](#)」を参照してください。

テストまたはデバッグを行う場合は、Jaeger が特定の操作を追跡できるように強制できます。詳細は、[Jaeger Python クライアントのドキュメント](#) を参照してください。

アプリケーションをキャプチャーするトレースを表示するには、[Jaeger Getting Started](#) を使用して Jaeger インフラストラクチャーおよびコンソールを実行します。

第13章 ファイルベースの設定

AMQ Python は、**connect.json** という名前のローカルファイルから接続を確立するために使用される設定オプションを読み取りできます。これにより、デプロイメント時にアプリケーションで接続を設定できます。

ライブラリーは、接続オプションを指定せずにアプリケーションがコンテナの **connect** メソッドを呼び出すと、ファイルの読み取りを試みます。

13.1. ファイルの場所

設定された場合、AMQ Python は **MESSAGING_CONNECT_FILE** 環境変数の値を使用して設定ファイルを見つけます。

MESSAGING_CONNECT_FILE が設定されていない場合、AMQ Python は以下の場所で **connect.json** という名前のファイルを検索します。最初の一致で停止します。

Linux の場合:

1. **\$PWD/connect.json**。ここでの **\$PWD** は、クライアントプロセスの現在の作業ディレクトリーです。
2. **\$HOME/.config/messaging/connect.json** **\$HOME** は、現在のユーザーのホームディレクトリーに置き換えます。
3. **/etc/messaging/connect.json**

Windows の場合:

1. **%cd%/connect.json**。ここでの **%cd%** は、クライアントプロセスの現在の作業ディレクトリーです。

connect.json ファイルが見つからない場合、ライブラリーはすべてのオプションにデフォルト値を使用します。

13.2. ファイルフォーマット

connect.json ファイルには JSON データが含まれ、JavaScript コメントの追加サポートが提供されません。

すべての設定属性は任意で、またはデフォルト値を持っているため、簡単な例にはいくつかの詳細のみが必要になります。

例: 簡単な **connect.json** ファイル

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

SASL および SSL/TLS オプションは、**"sasl"** および **"tls"** namespace で入れ子になっています。

例: SASL および SSL/TLS オプションを持つ **connect.json** ファイル

```

{
  "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. SSL 設定オプション

オプションキーは、ドット (.) を含む属性は namespace 内でネストされた属性を表します。

表13.1 connect.json の設定オプション

キー	値のタイプ	デフォルト値	説明
scheme	string	"amqps"	クリアテキストの "amqp" SSL/TLS の "amqps" の場合
host	string	"localhost"	リモートホストのホスト名または IP アドレス
port	文字列または番号	"amqps"	ポート番号またはポートリテラル
user	string	なし	認証のユーザー名
password	string	なし	認証のパスワード
sasl.mechanism s	リストまたは文字列	none (システムのデフォルト)	有効な SASL メカニズムの JSON リスト。ベア文字列は1つのメカニズムを表します。指定のない場合は、クライアントはシステムによって提供されるデフォルトのメカニズムを使用します。
sasl.allow_insecure	boolean	false	クリアテキストパスワードを送信するメカニズムの有効化
tls.cert	string	なし	クライアント証明書のファイル名またはデータベース ID
tls.key	string	なし	クライアント証明書の秘密鍵のファイル名またはデータベース ID
tls.ca	string	なし	CA 証明書のファイル名、ディレクトリー、またはデータベース ID

キー	値のタイプ	デフォルト値	説明
tls.verify	boolean	true	一致するホスト名を持つ有効なサーバー証明書が必要

第14章 相互運用性

本章では、AMQ Python を他の AMQ コンポーネントと組み合わせて使用方法を説明します。AMQ コンポーネントの互換性の概要は、「[製品の概要](#)」を参照してください。

14.1. 他の AMQP クライアントとの相互運用

AMQP メッセージは [AMQP タイプシステム](#) を使用して構成されます。この一般的な形式を使用するのは、異なる言語の AMQP クライアントが、相互運用できることが理由です。

メッセージを送信する場合、AMQ Python は自動的に言語ネイティブの型を AMQP エンコードデータに変換します。メッセージの受信時に、リバース変換が行われます。



注記

AMQP のタイプに関する詳細は、Apache Qpid プロジェクトによって維持される [インタラクティブタイプリファレンス](#) を参照してください。

表14.1 AMQP 型

AMQP 型	説明
null	空の値
boolean	true または false の値
char	単一の Unicode 文字
string	Unicode 文字のシーケンス
binary	バイト数のシーケンス
byte	署名済み 8 ビットの整数
short	署名済み 16 ビット整数
int	署名付き 32 ビット整数
long	署名済み 64 ビット整数
ubyte	署名なし 8 ビット整数
ushort	未署名の 16 ビット整数
uint	署名のない 32 ビット整数
ulong	未署名の 64 ビット整数
float	32 ビット浮動小数点数

AMQP 型	説明
double	64 ビット浮動小数点数
array	単一タイプの値シーケンス
list	変数タイプの値シーケンス
map	異なるキーから値へのマッピング
uuid	ユニバーサル一意識別子
symbol	制限されたドメインからの 7 ビットの ASCII 文字列
timestamp	絶対ポイント (時間単位)

表14.2 エンコード前およびデコード後における AMQ Python タイプ

AMQP 型	エンコード前の AMQ Python タイプ	デコード後の AMQ Python タイプ
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 型	エンコード前の AMQ Python タイプ	デコード後の AMQ Python タイプ
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

表14.3 AMQ Python およびその他の AMQ クライアントタイプ (1/2)

エンコード前の AMQ Python タイプ	AMQ C++ タイプ	AMQ JavaScript タイプ
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

エンコード前の AMQ Python タイプ	AMQ C++ タイプ	AMQ JavaScript タイプ
<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>

表14.4 AMQ Python およびその他の AMQ クライアントタイプ (2/2)

エンコード前の AMQ Python タイプ	AMQ .NET タイプ	AMQ Ruby タイプ
<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>

エンコード前の AMQ Python タイプ	AMQ .NET タイプ	AMQ Ruby タイプ
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. AMQ JMS での相互運用

AMQP は、JMS メッセージングモデルへの標準的なマッピングを定義します。本項では、そのマッピングのさまざまな側面について説明します。詳細は、「AMQ JMS [相互運用性](#)」を参照してください。

JMS メッセージタイプ

AMQ Python は、本文タイプが異なる、単一のメッセージを提供します。一方、JMS API は異なるメッセージタイプを使用して、さまざまな種類のデータを表します。以下の表は、特定のボディ型が JMS メッセージタイプにマッピングする方法を示しています。

結果として生成される JMS メッセージタイプの明示的な制御を行うために、`x-opt-jms-msg-type` メッセージアノテーションを設定できます。詳細は、「AMQ JMS [相互運用性](#)」の章を参照してください。

表14.5 AMQ Python および JMS メッセージタイプ

AMQ Python ボディのタイプ	JMS メッセージタイプ
unicode	TextMessage
None	TextMessage
bytes	BytesMessage
それ以外のタイプ	ObjectMessage

14.3. AMQ BROKER への接続

AMQ Broker は AMQP 1.0 クライアントと相互運用するために設計されています。以下をチェックして、ブローカーが AMQP メッセージング用に設定されていることを確認します。

- ネットワークファイアウォールのポート 5672 が開いている。
- AMQ Broker AMQP アクセプターが有効になっています。「[デフォルトのアクセプター設定](#)」を参照してください。
- 必要なアドレスはブローカーで設定されます。「[Addresses, Queues, and Topics](#)」を参照してください。
- ブローカーはクライアントからアクセスを許可するよう設定され、クライアントは必要なクレデンシャルを送信するよう設定されます。[Broker Security](#) を参照してください。

14.4. AMQ INTERCONNECT への接続

AMQ Interconnect は AMQP 1.0 クライアントと動作します。以下をチェックして、コンポーネントが正しく設定されていることを確認します。

- ネットワークファイアウォールのポート 5672 が開いている。
- ルーターはクライアントからアクセスを許可するよう設定され、クライアントは必要なクレデンシャルを送信するよう設定されます。「[ネットワーク接続のセキュリティー保護](#)」を参照してください。

付録A サブスクリプションの使用

AMQ は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

A.1. アカウントへのアクセス

手順

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

A.2. サブスクリプションのアクティベート

手順

1. access.redhat.com に移動します。
2. サブスクリプション に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

A.3. リリースファイルのダウンロード

.zip、.tar.gz、およびその他のリリースファイルにアクセスするには、カスタマーポータルを使用してダウンロードする関連ファイルを検索します。RPM パッケージまたは Red Hat Maven リポジトリを使用している場合、この手順は必要ありません。

手順

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **INTEGRATION AND AUTOMATION** カテゴリで **Red Hat AMQ** エントリーを見つけます。
3. 必要な AMQ 製品を選択します。 **Software Downloads** ページが開きます。
4. コンポーネントの **Download** リンクをクリックします。

A.4. パッケージを受信するためのシステムの登録

この製品の RPM パッケージを Red Hat Enterprise Linux にインストールするには、お使いのシステムを登録する必要があります。ダウンロードしたリリースファイルを使用している場合は、この手順は必要ありません。

手順

1. access.redhat.com に移動します。
2. **Registration Assistant** に移動します。

3. ご使用の OS バージョンを選択し、次のページに進みます。
4. システムの端末に一覧表示されたコマンドを使用して、登録を完了します。

システムを登録する方法は、以下のリソースを参照してください。

- [Red Hat Enterprise Linux 7 - システム登録およびサブスクリプション管理](#)
- [Red Hat Enterprise Linux 8 - システム登録およびサブスクリプション管理](#)

付録B RED HAT ENTERPRISE LINUX パッケージの使用

本セクションでは、Red Hat Enterprise Linux の RPM パッケージとして配信されるソフトウェアを使用する方法を説明します。

この製品の RPM パッケージが利用できるようにするには、最初に [システムを登録する](#) 必要があります。

B.1. 概要

ライブラリーやサーバーなどのコンポーネントには多くの場合、複数のパッケージが関連付けられています。それらをインストールする必要はありません。必要なものをインストールできます。

プライマリーパッケージには、通常、追加の修飾子がない最も単純な名前があります。このパッケージは、プログラムのランタイム時にコンポーネントを使用するために必要なすべてのインターフェースを提供します。

-devel で終わる名前を持つパッケージには、C ライブラリーおよび C++ ライブラリーのヘッダーが含まれます。これは、このパッケージに依存するプログラムを構築するためにコンパイル時に必要になります。

-docs に末尾の名前を持つパッケージには、コンポーネントのドキュメントおよびサンプルプログラムが含まれます。

RPM パッケージの使用方法は、以下のいずれかの資料を参照してください。

- [Red Hat Enterprise Linux 7: ソフトウェアのインストールおよび管理](#)
- [Red Hat Enterprise Linux 8 - ソフトウェアパッケージの管理](#)

B.2. パッケージの検索

パッケージを検索するには、**yum search** コマンドを使用します。検索結果にはパッケージ名が含まれます。パッケージ名は、このセクションに記載されている他のコマンドで **<package>** の値として使用できます。

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

B.3. パッケージのインストール

パッケージをインストールするには、**yum install** コマンドを使用します。

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

B.4. パッケージ情報のクエリー

システムにインストールされているパッケージを一覧表示するには、**rpm -qa** コマンドを使用します。

```
$ rpm -qa
```

特定のパッケージに関する情報を取得するには、**rpm -qi** コマンドを使用します。

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

パッケージに関連するファイルを一覧表示するには、**rpm -qi** コマンドを使用します。

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


C.4. ブローカーの停止

サンプルの実行が終了したら、**artemis stop** コマンドを使用してブローカーを停止します。

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

改訂日時: 2021-08-29 15:58:03 +1000