Red Hat AMQ 7.7

AMQ C++ クライアントの使用

AMQ Clients 2.7 向け

Last Updated: 2023-01-29

Red Hat AMQ 7.7 AMQ C++ クライアントの使用

AMQ Clients 2.7 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 © 2023 | You need to change the HOLDER entity in the en-US/Using_the_AMQ_Cpp_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、クライアントをインストールして設定する方法、実例を実行し、他の AMQ コンポーネントでクライアントを使用する方法を説明します。

目次

第1章 概要 4
1.1. 主な特長 4
1.2. サポートされる標準およびプロトコル 4
1.3. サポートされる構成 4
1.4. 用語および概念 5
1.5. 本書の表記慣例 6
sudo コマンド 6
ファイルパス 6
変数テキスト 6

第2章 インストール 7
2.1. 前提条件 7
2.2. RED HAT ENTERPRISE LINUX へのインストール を参照してください。 7
2.3. MICROSOFT WINDOWS へのインストール を参照してください。 7

第3章 スタートガイド 9
3.1. 前提条件 9
3.2. RED HAT ENTERPRISE LINUX での HELLO WORLD の実行 9

第4章 例 10
4.1. メッセージの送信 10
サンプルの実行 11
4.2. メッセージの受信 11
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. コンテナーの作成 14
5.3. コンテナーアイデンティティーの設定 14

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. 内向き接続の許可 18

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 送信者と受信者 21
8.1. オンデマンドでの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. 例外の処理 26
10.2. 接続およびプロトコルエラーの処理 26
10.3. ロギング 27
10.3.1. プロトコルロギングの有効化 27

第11章 スレッドおよびスケジューリング 28
11.1. スレッドモデル 28
11.2. スレッドセーフルール 28
11.3. ワークキュー 28
11.4. ウェイクプリミティブ 28
11.5. スケジュールが遅延しているワーク 29
11.6. 古いバージョンの C++ の使用 29

第12章 ファイルベースの設定 31
12.1. ファイルの場所 31
12.2. ファイル形式 31
12.3. 設定オプション 32

第13章 相互運用性 33
13.1. 他の AMQP クライアントとの相互運用 33
13.2. AMQ JMS での相互運用 37
JMS メッセージタイプ 37
13.3. AMQ BROKER への接続 37
13.4. 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. パッケージ用のシステムの登録 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 例で AMQ ブローカーの使用 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

第1章 概要

AMQ C++ 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。また、AMQP メッセージを送受信する C++ アプリケーションを作成できます。

AMQ C++ は AMQ Clients (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) に含まれています。クライアントの概要は、AMQ Clients の概要 を参照してください。本リリースに関する詳細は、AMQ Clients 2.7 リリースノート を参照してください。

AMQ C++ 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 C++ 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 と言語ネイティブのデータ型間のシームレスな変換
- AMQP 1.0 の全機能へのアクセス

1.2. サポートされる標準およびプロトコル

AMQ C++ 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 C++ は、次に示す OS と言語のバージョンをサポートしています。詳細は、Red Hat AMQ 7 Supported Configurations を参照してください。

- Red Hat Enterprise Linux 6 with GNU C++、C++03 または C++0x としてコンパイル (部分的な C++11 サポート)
- GNU C++ を使用する Red Hat Enterprise Linux 7 および 8、C++03 または C++11 としてコンパイル
- Microsoft Visual Studio 2015 以降を搭載した Microsoft Windows 10 Pro
- Microsoft Visual Studio 2015 以降を搭載した Microsoft Windows Server 2012 R2 および 2016

AMQ C++ は、次の AMQ コンポーネントおよびバージョンとの組み合わせでサポートされています。

- AMQ ブローカーのすべてのバージョン
- AMQ Interconnect のすべてのバージョン
- AMQ Online のすべてのバージョン
- A-MQ 6 バージョン 6.2.1 以降

1.4. 用語および概念

本セクションでは、コア API エンティティーを紹介し、コア API が連携する方法を説明します。

表1.1 API の用語

エンティティー	説明
Container	接続の最上位のコンテナー。
接続	ネットワーク上の 2 つのピア間の通信チャネル。これにはセッションが含まれます。
Session	メッセージの送受信を行うためのコンテキスト。送信者および受信者が含まれます。
sender	メッセージをターゲットに送信するためのチャネル。これにはターゲットがあります。
receiver	ソースからメッセージを受信するためのチャネル。これにはソースがあります。
Source	メッセージの名前付きの発信元。
Target	メッセージの名前付き受信先。
メッセージ	情報のアプリケーション固有の部分。
Delivery	メッセージの転送。

AMQ C++ は **メッセージ** を送受信します。メッセージは、**senders** と **receivers** を介して、接続されたピアの間で転送されます。送信側および受信側は **セッション** 上で確立されます。セッションは**接続**上で確立されます。接続は、一意に識別された 2 つの **コンテナー** 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、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、メッセージ送信用の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、メッセージ受信用の受信者を作成します。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。メッセージとは、送信される内容のことで、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チェンジです。

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が解決します。送信側または受信側が解決されたことを知らせると、その配信の通信ができなくなります。受信側は、メッセージを受諾するか、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 権限を必要とするすべてのコマンドに対して **sudo** が使用されています。すべて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
本書では、変数を含むコードブロックが紹介されていますが、これは、お客様の環境に固有の値に置き換える必要があります。可変テキストは矢印の中括弧で囲まれ、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、以下のコマンドでは **<project-dir>** は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 C++ 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ーにアクセスするには、サブスクリプション が必要です。
- パッケージを Red Hat Enterprise Linux にインストールするには、システムが登録されている必要があります。
- Red Hat Enterprise Linux で AMQ C++ を使用してプログラムを構築するには、**gcc-c++** パッケージ、**cmake** パッケージ、および **make** パッケージをインストールする必要があります。
- Microsoft Windows で AMQ C++ を使用してプログラムを構築するには、Visual Studio をインストールする必要があります。

2.2. RED HAT ENTERPRISE LINUX へのインストール 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ーをサブスクライブします。必要に応じて、**<variant>** を Red Hat Enterprise Linux のバリアントの値 (例えば、**server** または **workstation**) に置き換えます。

   Red Hat Enterprise Linux 6

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

   Red Hat Enterprise Linux 7

   ```
   $ sudo subscription-manager repos --enable=amq-clients-2-for-rhel-7-<variant>-rpms
   ```

   Red Hat Enterprise Linux 8

   ```
   $ sudo subscription-manager repos --enable=amq-clients-2-for-rhel-8-x86_64-rpms
   ```

2. **yum** コマンドを使用して、**qpid-proton-cpp-devel** パッケージおよび **qpid-proton-cpp-docs** パッケージをインストールします。

   ```
   $ sudo yum install qpid-proton-cpp-devel qpid-proton-cpp-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. MICROSOFT WINDOWS へのインストール 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** カテゴリーで **Red Hat AMQ Clients** 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. **AMQ Clients 2.7.0 C++**.zip ファイルをダウンロードします。
5. zip ファイルを右クリックし、**Extract All** を選択して、選択したディレクトリーにファイルの内容を展開します。

.zip ファイルの内容を抽出すると、**amq-clients-2.7.0-cpp-win** という名前のディレクトリーが作成されます。これはインストールの最上位ディレクトリーであり、本書では **<install-dir>** と呼びます。

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

- ご使用の環境のインストール手順を完了する必要があります。
- インターフェイス **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、グリーティングを含むメッセージを**examples**キューに送信して、受信しなおします。成功すると、受信したメッセージをコンソールに出力します。

手順

1. サンプルを選択した場所にコピーします。

```
$ cp -r /usr/share/proton/examples/cpp cpp-examples
```

2. ビルドディレクトリーを作成し、そのディレクトリーに移動します。

```
$ mkdir cpp-examples/bld
$ cd cpp-examples/bld
```

3. **cmake** を使用してビルドを設定し、**make** を使用して例をコンパイルします。

```
$ cmake ..
$ make
```

4. **helloworld** プログラムを実行します。

```
$ ./helloworld
Hello World!
```

第4章 例

本章では、サンプルプログラムで AMQ C++ を使用する方法について説明します。

その他の例については、AMQ C++ サンプルスイート を参照してください。

注記

本ガイドのコードは、C++11 機能を使用します。AMQ C++ は C++03 とも互換性がありますが、コードには若干の変更が必要です。

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、ターゲット **<address>** の送信者を作成し、**<message-body>** を含むメッセージを送信して接続を切断して終了します。

例: メッセージの送信

```
#include <proton/connection.hpp>
#include <proton/container.hpp>
#include <proton/message.hpp>
#include <proton/messaging_handler.hpp>
#include <proton/sender.hpp>
#include <proton/target.hpp>

#include <iostream>
#include <string>

struct send_handler : public proton::messaging_handler {
    std::string conn_url_ {};
    std::string address_ {};
    std::string message_body_ {};

    void on_container_start(proton::container& cont) override {
        cont.connect(conn_url_);

        // To connect with a user and password:
        //
        // proton::connection_options opts {};
        // opts.user("<user>");
        // opts.password("<password>");
        //
        // cont.connect(conn_url_, opts);
    }

    void on_connection_open(proton::connection& conn) override {
        conn.open_sender(address_);
    }

    void on_sender_open(proton::sender& snd) override {
        std::cout << "SEND: Opened sender for target address '"
                  << snd.target().address() << "'\n";
    }
```

```
    void on_sendable(proton::sender& snd) override {
        proton::message msg {message_body_};
        snd.send(msg);

        std::cout << "SEND: Sent message '" << msg.body() << "'\n";

        snd.close();
        snd.connection().close();
    }
};

int main(int argc, char** argv) {
    if (argc != 4) {
        std::cerr << "Usage: send <connection-url> <address> <message-body>\n";
        return 1;
    }

    send_handler handler {};
    handler.conn_url_ = argv[1];
    handler.address_ = argv[2];
    handler.message_body_ = argv[3];

    proton::container cont {handler};

    try {
        cont.run();
    } catch (const std::exception& e) {
        std::cerr << e.what() << "\n";
        return 1;
    }

    return 0;
}
```

サンプルの実行

サンプルプログラムを実行するには、ローカルファイルにコピーしてコンパイルし、コマンドラインから実行します。

```
$ g++ send.cpp -o send -std=c++11 -lstdc++ -lqpid-proton-cpp
$ ./send 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>** の受信側を作成し、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

```
#include <proton/connection.hpp>
#include <proton/container.hpp>
#include <proton/delivery.hpp>
#include <proton/message.hpp>
#include <proton/messaging_handler.hpp>
#include <proton/receiver.hpp>
```

```
#include <proton/source.hpp>

#include <iostream>
#include <string>

struct receive_handler : public proton::messaging_handler {
    std::string conn_url_ {};
    std::string address_ {};
    int desired_ {0};
    int received_ {0};

    void on_container_start(proton::container& cont) override {
        cont.connect(conn_url_);

        // To connect with a user and password:
        //
        // proton::connection_options opts {};
        // opts.user("<user>");
        // opts.password("<password>");
        //
        // cont.connect(conn_url_, opts);
    }

    void on_connection_open(proton::connection& conn) override {
        conn.open_receiver(address_);
    }

    void on_receiver_open(proton::receiver& rcv) override {
        std::cout << "RECEIVE: Opened receiver for source address '"
                  << rcv.source().address() << "'\n";
    }

    void on_message(proton::delivery& dlv, proton::message& msg) override {
        std::cout << "RECEIVE: Received message '" << msg.body() << "'\n";

        received_++;

        if (received_ == desired_) {
            dlv.receiver().close();
            dlv.connection().close();
        }
    }
};

int main(int argc, char** argv) {
    if (argc != 3 && argc != 4) {
        std::cerr << "Usage: receive <connection-url> <address> [<message-count>]\n";
        return 1;
    }

    receive_handler handler {};
    handler.conn_url_ = argv[1];
    handler.address_ = argv[2];

    if (argc == 4) {
        handler.desired_ = std::stoi(argv[3]);
```

```
    }

    proton::container cont {handler};

    try {
        cont.run();
    } catch (const std::exception& e) {
        std::cerr << e.what() << "\n";
        return 1;
    }

    return 0;
}
```

サンプルの実行

サンプルプログラムを実行するには、ローカルファイルにコピーしてコンパイルし、コマンドラインから実行します。

```
$ g++ receive.cpp -o receive -std=c++11 -lstdc++ -lqpid-proton-cpp
$ ./receive amqp://localhost queue1
```

第5章 API の使用

詳細は、 AMQ C++ API reference および AMQ C++ example suite 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 C++ 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 **messaging_handler** クラスでコールバックメソッドを実装します。これらの方法は、ネットワークアクティビティーとして呼び出され、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

```
struct example_handler : public proton::messaging_handler {
    void on_container_start(proton::container& cont) override {
        std::cout << "The container has started\n";
    }

    void on_sendable(proton::sender& snd) override {
        std::cout << "A message can be sent\n";
    }

    void on_message(proton::delivery& dlv, proton::message& msg) override {
        std::cout << "A message is received\n";
    }
};
```

これらはごく一部の一般的なケースイベントのみです。全セットは AMQ API リファレンス に文書化されています。

5.2. コンテナーの作成

コンテナーは最上位の 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: コンテナーの作成

```
int main() {
    example_handler handler {};
    proton::container cont {handler};
    cont.run();
}
```

5.3. コンテナーアイデンティティーの設定

各コンテナーインスタンスには、コンテナー ID と呼ばれる一意のアイデンティティーがあります。AMQ C++ がネットワーク接続を作成する場合は、コンテナー ID をリモートピアに送信します。コンテナー 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: コンテナーアイデンティティーの設定

```
proton::container cont {handler, "job-processor-3"};
```

ユーザーが ID を設定しない場合には、コンテナーが処理されると、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

接続 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 暗号化のある 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。値は、ホスト名または数値の IP アドレスの場合があります。IPv6 アドレスは角括弧で囲む必要があります。
- **ポート** - 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 サンプル

```
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 で **container::connect()** メソッドを呼び出します。通常、これは **messaging_handler::on_container_start()** メソッド内で行われます。

例: 外向き接続の作成

```
class example_handler : public proton::messaging_handler {
    void on_container_start(proton::container& cont) override {
        cont.connect("amqp://example.com");
    }

    void on_connection_open(proton::connection& conn) override {
        std::cout << "The connection is open\n";
    }
};
```

セキュアな接続の作成については、*7章 セキュリティー* セクションを参照してください。

6.3. 再接続の設定

再接続することで、クライアントは失われた接続から復旧できます。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るように使用されます。

AMQ C++ はデフォルトで再接続を無効にします。これを有効にするには、**reconnect** 接続オプションを **reconnect_options** クラスのインスタンスに設定します。

例: 再接続の有効化

```
proton::connection_options opts {};
proton::reconnect_options ropts {};

opts.reconnect(ropts);

container.connect("amqp://example.com", opts);
```

再接続を有効にすると、接続が失われた場合、または接続の試行が失敗した場合に、クライアントは少し遅れて再試行します。遅延は新規試行ごとに指数関数的に増加します。

接続試行間の遅延を制御するには、**delay** オプション、**delay_multiplier** オプション、および **max_delay** オプションを設定します。すべての期間はミリ秒単位で指定します。

再接続試行回数を制限するには、**max_attempts** オプションを設定します。これを 0 に設定すると制限が削除されます。

例: 再接続の設定例

```
proton::connection_options opts {};
proton::reconnect_options ropts {};

ropts.delay(proton::duration(10));
ropts.delay_multiplier(2.0);
ropts.max_delay(proton::duration::FOREVER);
ropts.max_attempts(0);

opts.reconnect(ropts);

container.connect("amqp://example.com", opts);
```

6.4. フェイルオーバーの設定

AMQ C++ 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。一覧が使い切られると、プロセスは最初から開始します。

別の接続エンドポイントを指定するには、**failover_urls** reconnect 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

```
std::vector<std::string> failover_urls = {
    "amqp://backup1.example.com",
    "amqp://backup2.example.com"
};

proton::connection_options opts {};
proton::reconnect_options ropts {};

opts.reconnect(ropts);
ropts.failover_urls(failover_urls);

container.connect("amqp://primary.example.com", opts);
```

6.5. 内向き接続の許可

AMQ C++ 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、**proton::container::listen()** メソッドを使用して、ローカルホストアドレスとリッスンするポートが含まれる URL を指定します。

例: 内向き接続の許可

```
class example_handler : public proton::messaging_handler {
    void on_container_start(proton::container& cont) override {
        cont.listen("0.0.0.0");
    }

    void on_connection_open(proton::connection& conn) override {
        std::cout << "New incoming connection\n";
    }
};
```

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.cpp 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 C++ 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、**ssl_client_options** 接続オプションを設定し、**amqps** スキームで接続 URL を使用します。 **ssl_client_options** コンストラクターは、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 を取得します。

例: SSL/TLS の有効化

```
proton::ssl_client_options sopts {"/etc/pki/ca-trust"};
proton::connection_options opts {};

opts.ssl_client_options(sopts);

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

7.2. ユーザーとパスワードを使用した接続

AMQ C++ 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** メソッドに **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

```
proton::connection_options opts {};

opts.user("alice");
opts.password("secret");

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

7.3. SASL 認証の設定

AMQ C++ は SASL プロトコルを使用して認証を実行します。SASL はさまざまな認証 **メカニズム** を使用できます。2 つのネットワークピアが接続すると、許可されたメカニズムが交換され、両方で許可されてい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 C++ 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、**sasl_allowed_mechs** 接続オプションを使用します。このオプションは、スペースで区切られ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

```
proton::connection_options opts {};

opts.sasl_allowed_mechs("ANONYMOUS");

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

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**、**EXTERNAL** が含まれます。

AMQ C++ はデフォルトで再接続を有効にします。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

```
proton::connection_options opts {};

opts.sasl_enabled(false);

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

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. 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 を設定するには 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。

   ```
   proton::connection_options opts {};

   opts.sasl_allowed_mechs("GSSAPI");

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

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 送信者と受信者

クライアントは、送信者と受信者のリンクを使用して、メッセージ配信のチャネルを表現します。送信者と受信者は一方向であり、送信元はメッセージの発信元に、ターゲットはメッセージの宛先になります。

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. オンデマンドでのキューとトピックの作成

メッセージサーバーによっては、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。送信側または受信側が割り当てられている場合、サーバーは送信側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で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用) 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** 機能を設定してどちらを優先するかを示すことができます。

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. キューとトピックを自動的に作成するようにメッセージサーバーを設定します。多くの場合、これがデフォルト設定になります。

2. 以下の例のように、送信者ターゲットまたは受信者ソースに **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

```
void on_container_start(proton::container& cont) override {
    proton::connection conn = cont.connect("amqp://example.com");
    proton::sender_options opts {};
    proton::target_options topts {};

    topts.capabilities(std::vector<proton::symbol> { "queue" });
    opts.target(topts);

    conn.open_sender("jobs", opts);
}
```

例: オンデマンドで作成されたトピックからの受信

```
void on_container_start(proton::container& cont) override {
    proton::connection conn = cont.connect("amqp://example.com");
    proton::receiver_options opts {};
    proton::source_options sopts {};

    sopts.capabilities(std::vector<proton::symbol> { "topic" });
    opts.source(sopts);

    conn.open_receiver("notifications", opts);
}
```

詳細は、以下の例を参照してください。

- queue-send.cpp
- queue-receive.cpp
- topic-send.cpp
- topic-receive.cpp

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、メッセージ受信者は、クライアントが終了すると、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、クライアントはそれらのサブスクリプションの割り当てを解除してから、後で再度アタッチできます。デタッチ時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ー ID とレシーバー名を組み合わせて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。これらには、サブスクリプションを回復できるように、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. 接続コンテナー ID を **client-1** などの安定した値に設定します。

```
proton::container cont {handler, "client-1"};
```

2. **sub-1** などの安定した名前で受信側を作成し、**durability_mode** および **expiry_policy** プロパティーを指定して、受信者のソースが永続化されるように設定します。

```
void on_container_start(proton::container& cont) override {
  proton::connection conn = cont.connect("amqp://example.com");
  proton::receiver_options opts {};
  proton::source_options sopts {};

  opts.name("sub-1");
  sopts.durability_mode(proton::source::UNSETTLED_STATE);
  sopts.expiry_policy(proton::source::NEVER);

  opts.source(sopts);

  conn.open_receiver("notifications", opts);
}
```

サブスクリプションからデタッチするには、**proton::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、**proton::receiver::close()** メソッドを使用します。

詳細は、subscribe.cpp 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。このサブスクリプションは共有されているため、複数のクライアントが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、受信者のソースに**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ー ID とレシーバー名を組み合わせて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特定できるように、これらに安定した値を指定する必要があります。**shared** に加えて **global** 機能が設定されている場合、サブスクリプション識別に受信者名だけが使用されます。

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

```
proton::container cont {handler, "client-1"};
```

2. **sub-1** などの安定した名前で受信者を作成し、**shared** 機能 を設定して共有用に受信者のソースを設定します。

```
void on_container_start(proton::container& cont) override {
    proton::connection conn = cont.connect("amqp://example.com");
    proton::receiver_options opts {};
    proton::source_options sopts {};

    opts.name("sub-1");
    sopts.capabilities(std::vector<proton::symbol> { "shared" });

    opts.source(sopts);

    conn.open_receiver("notifications", opts);
}
```

サブスクリプションからデタッチするには、**proton::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、**proton::receiver::close()** メソッドを使用します。

詳細は、shared-subscribe.cpp 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** なイベントは、**proton::sender** に少なくとも 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場合に実行されます。

例: メッセージの送信

```
struct example_handler : public proton::messaging_handler {
    void on_container_start(proton::container& cont) override {
        proton::connection conn = cont.connect("amqp://example.com");
        conn.open_sender("jobs");
    }

    void on_sendable(proton::sender& snd) override {
        proton::message msg {"job-1"};
        snd.send(msg);
    }
};
```

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場合、送信側は転送を表す **tracker** オブジェクトへの参照を保持することができます。受信側は、配信される各メッセージを受諾または拒否します。追跡された各配信の結果の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_tracker_accept** イベントハンドラーおよび **on_tracker_reject** イベントハンドラーを上書きし、配信状態の更新を **send()** から返されたトラッカーにマップします。

例: 送信したメッセージの追跡

```
void on_sendable(proton::sender& snd) override {
    proton::message msg {"job-1"};
    proton::tracker trk = snd.send(msg);
}

void on_tracker_accept(proton::tracker& trk) override {
    std::cout << "Delivery for " << trk << " is accepted\n";
}

void on_tracker_reject(proton::tracker& trk) override {
    std::cout << "Delivery for " << trk << " is rejected\n";
}
```

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

-

```
struct example_handler : public proton::messaging_handler {
    void on_container_start(proton::container& cont) override {
        proton::connection conn = cont.connect("amqp://example.com");
        conn.open_receiver("jobs");
    }

    void on_message(proton::delivery& dlv, proton::message& msg) override {
        std::cout << "Received message '" << msg.body() << "'\n";
    }
};
```

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 **delivery::accept ()** または **delivery::reject()** メソッドを使用します。

例: 受信したメッセージの承認

```
void on_message(proton::delivery& dlv, proton::message& msg) override {
    try {
        process_message(msg);
        dlv.accept();
    } catch (std::exception& e) {
        dlv.reject();
    }
}
```

デフォルトでは、配信を明示的に確認しないと、ライブラリーは **on_message** が返された後に受け入れます。この動作を無効にするには、**auto_accept** receiver オプションを false に設定します。

第10章 エラー処理

AMQ C++ 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処理するための仮想関数のオーバーライド

例外の取得は最も基本的なものですが、エラーを処理する詳細な方法です。ハンドラールーチンでオーバーライドを使用してエラーが処理されない場合、例外が出力され、キャッチして処理することができます。

10.1. 例外の処理

ハンドラールーチンでオーバーライドを使用してエラーが処理されない場合、コンテナーの **run** メソッドによって例外が出力されます。

AMQ C++ が **proton::error** クラスから継承するすべての例外。これにより、 **std::runtime_error** クラスおよび **std:: exception** クラスが継承さ れます。

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

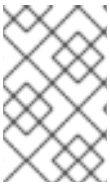
例: API 固有の例外処理

```
try {
    // Something that might throw an exception
} catch (proton::error& e) {
    // Handle Proton-specific problems here
} catch (std::exception& e) {
    // Handle more general problems here
}
```

API 固有の例外処理が必要ない場合は、**proton::error** が継承されるため 、**std::exception** のみをキャッチする必要があります。

例: 一般的な例外処理

```
int main() {
    try {
        // Something that might throw an exception
    } catch (std::exception& e) {
        std::cerr << "Caught exception: " << e.what() << std::endl;
    }
}
```



注記

C++ プログラムのすべての例外は **std::exception** から継承されるため、コードブロックを記述して **main** メソッドをラップし、**std::exception** エラーに関する情報を表示できます。

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できます。

- **on_transport_error(proton::transport&)**
- **on_connection_error(proton::connection&)**
- **on_session_error(proton::session&)**
- **on_receiver_error(proton::receiver&)**
- **on_sender_error(proton::sender&)**

これらのイベント処理ルーチンは、イベント内の特定のオブジェクトにエラー状態が発生す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。

オーバーライドされない場合、デフォルトのエラーハンドラーが呼び出され、発生したエラー状態が示されます。

特定のエラーハンドラーのいずれかが上書きされない場合は、デフォルトのエラーハンドラーが呼び出されます。

- **on_error(proton::error_condition&)**

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。リソースのクリーンアップは、近辺にある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は、通常は、一般的な **on_error** ハンドラーを使用してより具体的なハンドラーを用意しません。

10.3. ロギング

10.3.1. 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。多くの場合、このデータは問題の診断時に重要になります。

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11章 スレッドおよびスケジューリング

AMQ C++ は、C++11 以降での完全なマルチスレッドをサポートします。古いバージョンの C++ では、限定されたマルチスレッドが可能です。「古いバージョンの C++ の使用」を参照してください。

11.1. スレッドモデル

container オブジェクトは複数の接続を同時に処理できます。AMQP イベントが接続で発生するとき、コンテナーは **messaging_handler** コールバック関数を呼び出します。1 つの接続のコールバックはシリアライズされます (同時に呼び出されません)。しかし、異なる接続のコールバックは並行して安全に実行できます。

ハンドラー接続オプションを使用して、**container::connect()** または **listen_handler::on_accept()** の接続に **handler** を割り当てることができます。ライブラリースレッドによる同時使用から保護するために、ハンドラーによるロックやその他の同期を必要としないように、各接続に個別のハンドラーを作成することが推奨されます。非ライブラリースレッドがハンドラーを同時に使用する場合は、同期が必要です。

11.2. スレッドセーフルール

connection、**session**、**sender**、**receiver**、**tracker**、および **delivery** オブジェクトはスレッドセーフではなく、以下のルールの対象となります。

1. これらは **messaging_handler** コールバックまたは **work_queue** 関数からのみ使用する必要があります。
2. 別のコネクションのコールバックからある接続に属するオブジェクトを使用することはできません。
3. ルール 2 を考慮した場合は、後のコールバックで使用するために AMQ C++ オブジェクトをメンバー変数に保存できます。

message オブジェクトは、標準の C++ の組み込みタイプと同じスレッド制約を持つ値タイプです。これは同時に変更することはできません。

11.3. ワークキュー

work_queue インターフェイスは、異なる接続ハンドラーまたは非ライブラリースレッドと接続ハンドラーとの間で安全な通信を行う方法を提供します。

- 各接続には、関連する **work_queue** があります。
- ワークキューはスレッドセーフ (C++11 以上) です。スレッドで作業を追加できます。
- **work** アイテムは **std::function** で、バインドされた引数はイベントコールバックのように呼び出されます。

ライブラリーがワーク関数を呼び出すと、イベントコールバックのように関数を処理し、ハンドラーおよび AMQ C++ オブジェクトに安全にアクセスできるように、安全に作業関数を処理できます。

11.4. ウェイクプリミティブ

connection::wake() メソッドを 使用すると、**on_connection_wake()** コールバックをトリガーして、スレッドが接続でアクティビティーを要求することが できます。これは、**connection** 上のスレッドセーフメソッドのみです。

wake() は、スレッド間をシグナル化する軽量な低レベルのプリミティブです。

- **work_queue** とは異なり、コードやデータは含まれません。
- **wake()** への複数の呼び出しは、単一の **on_connection_wake()** に結合される可能性があります。
- **on_connection_wake()** への呼び出しは、ライブラリーが内部的に **wake()** を使用するため、**wake()** へのアプリケーション呼び出しなしに発生する可能性があります。

wake() のセマンティクスは **std::condition_variable::notify_one()** に似ています。ウェイクアップが発生しますが、ウェイクアップが発生した理由とその動作 (ある場合) を判断するために、共有アプリケーションの状態が必要です。

ワークキューは多くのインスタンスで簡単に使用できますが、独自の外部スレッドセーフキューがあり、データを確認するための効率的な方法が必要な場合には、**wake ()** が役に立つことがあります。

11.5. スケジュールが遅延しているワーク

AMQ C++ には、遅延後にコードを実行する機能があります。これを使用して、定期的にスケジュールされた作業やタイムアウトなど、アプリケーションに時間ベースの動作を実装できます。

一定期間を遅らせるには、**schedule** メソッドを使用して遅延を設定し、作業を定義する関数を登録します。

例: 遅延後のメッセージの送信

```
void on_sender_open(proton::sender& snd) override {
    proton::duration interval {5 * proton::duration::SECOND};
    snd.work_queue().schedule(interval, [=] { send(snd); });
}

void send(proton::sender snd) {
    if (snd.credit() > 0) {
        proton::message msg {"hello"};
        snd.send(msg);
    }
}
```

この例では、送信者のワークキューで **schedule** メソッドを使用して、作業の実行コンテキストとして確立します。

11.6. 古いバージョンの C++ の使用

C++11 より前のバージョンでは、C++ のスレッドに対する標準サポートがありませんでした。スレッドに AMQ C++ を使用できますが、以下の制限があります。

- コンテナーはスレッドを作成しません。**container::run()** を呼び出す単一スレッドのみを使用します。

- **container** および **work_queue** を含む、AMQ C++ ライブラリークラスはいずれもスレッドセーフです。複数のスレッドで **container** を使用するには、外部ロックが必要です。唯一の例外は **connection::wake()** です。これは、古い C++ であってもスレッドセーフです。

container::schedule() および **work_queue** API は、C++11 lambda 関数を使用して作業単位を定義します。lambda をサポートしないバージョンの C++ を使用している場合は、代わりに **make_work()** 関数を使用する必要があります。

第12章 ファイルベースの設定

AMQ C++ は、**connect.json** という名前のローカルファイルからの接続確立に使用される設定オプションを読み取ることができます。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ーの **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。

12.1. ファイルの場所

設定されている場合には、AMQ C++ は **MESSAGING_CONNECT_FILE** 環境変数の値を使用して設定ファイルを検索します。

MESSAGING_CONNECT_FILE が設定されていない場合には、AMQ C++ 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. **$PWD/connect.json**: **$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**: **%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。

12.2. ファイル形式

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す。

設定属性はすべてオプションであるか、デフォルト値があるため、簡単な例では詳細をいくつか指定するだけで済みます。

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 オプションを含む 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"
    }
}
```

12.3. 設定オプション

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

表12.1 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"**	SSL/TLS のクリアテキストまたは **"amqps"** の場合は **"amqp"**
host	string	**"localhost"**	リモートホストのホスト名または IP アドレス
ポート	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	None	認証のユーザー名
password	string	None	認証のパスワード
sasl.mechanisms	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。指定がない場合、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	None	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	None	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	None	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
tls.verify	boolean	**true**	ホスト名が一致する、有効なサーバー証明書が必要

第13章 相互運用性

本章では、AMQ C++ を他の AMQ コンポーネントと組み合わせて使用する方法を説明します。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。

13.1. 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設定されます。このような一般的な形式は、異なる言語の AMQP クライアントが相互に対話できる理由の 1 つです。

メッセージを送信する場合、AMQ C++ は自動的に言語ネイティブの型を AMQP でエンコードされた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 タイプの詳細は、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13.1 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	バイトのシーケンス
byte	署名済み 8 ビット整数
short	署名済み 16 ビット整数
int	署名済み 32 ビット整数
long	署名済み 64 ビット整数
ubyte	署名なしの 8 ビット整数
ushort	署名なしの 16 ビット整数
uint	署名なしの 32 ビット整数
ulong	署名なしの 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	単一型の値シーケンス
list	変数型の値シーケンス
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	絶対的な時点

表13.2 エンコード前およびデコード後における AMQ C++ タイプ

AMQP 型	エンコード前の AMQ C++ タイプ	デコード後の AMQ C++ タイプ
null	**nullptr**	**nullptr**
boolean	**bool**	**bool**
char	**wchar_t**	**wchar_t**
string	**std::string**	**std::string**
binary	**proton::binary**	**proton::binary**
byte	**int8_t**	**int8_t**
short	**int16_t**	**int16_t**
int	**int32_t**	**int32_t**
long	**int64_t**	**int64_t**
ubyte	**uint8_t**	**uint8_t**
ushort	**uint16_t**	**uint16_t**
uint	**uint32_t**	**uint32_t**
ulong	**uint64_t**	**uint64_t**

AMQP 型	エンコード前の AMQ C++ タイプ	デコード後の AMQ C++ タイプ
float	float	float
double	double	double
list	std::vector	std::vector
map	std::map	std::map
uuid	proton::uuid	proton::uuid
symbol	proton::symbol	proton::symbol
timestamp	proton::timestamp	proton::timestamp

表13.3 AMQ C++ 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 C++ タイプ	AMQ JavaScript タイプ	AMQ .NET タイプ
nullptr	null	null
bool	boolean	System.Boolean
wchar_t	number	System.Char
std::string	string	System.String
proton::binary	string	System.Byte[]
int8_t	number	System.SByte
int16_t	number	System.Int16
int32_t	number	System.Int32
int64_t	number	System.Int64
uint8_t	number	System.Byte
uint16_t	number	System.UInt16
uint32_t	number	System.UInt32
uint64_t	number	System.UInt64

エンコード前の AMQ C++ タイプ	AMQ JavaScript タイプ	AMQ .NET タイプ
float	**number**	**System.Single**
double	**number**	**System.Double**
std::vector	**Array**	**Amqp.List**
std::map	**object**	**Amqp.Map**
proton::uuid	**number**	**System.Guid**
proton::symbol	**string**	**Amqp.Symbol**
proton::timestamp	**number**	**System.DateTime**

表13.4 AMQ C++ 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 C++ タイプ	AMQ Python タイプ	AMQ Ruby タイプ
nullptr	**None**	**nil**
bool	**bool**	**true, false**
wchar_t	**unicode**	**String**
std::string	**unicode**	**String**
proton::binary	**bytes**	**String**
int8_t	**int**	**Integer**
int16_t	**int**	**Integer**
int32_t	**long**	**Integer**
int64_t	**long**	**Integer**
uint8_t	**long**	**Integer**
uint16_t	**long**	**Integer**
uint32_t	**long**	**Integer**
uint64_t	**long**	**Integer**

エンコード前の AMQ C++ タイプ	AMQ Python タイプ	AMQ Ruby タイプ
float	**float**	**Float**
double	**float**	**Float**
std::vector	**list**	**Array**
std::map	**dict**	**Hash**
proton::uuid	-	-
proton::symbol	**str**	**Symbol**
proton::timestamp	**long**	**Time**

13.2. 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マッピングを定義します。本セクションでは、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 Interoperability の章を参照してください。

JMS メッセージタイプ

AMQ C++ 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。次の表は、特定の本文タイプが JMS メッセージタイプにどのようにマップされるかを示しています。

作成される JMS メッセージタイプをさらに明示的に制御するには、**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 Interoperability の章を参照してください。

表13.5 AMQ C++ および JMS メッセージタイプ

AMQ C++ ボディータイプ	JMS メッセージタイプ
std::string	**TextMessage**
nullptr	**TextMessage**
proton::binary	**BytesMessage**
それ以外のタイプ	**ObjectMessage**

13.3. 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。以下を確認して、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る。デフォルトのアクセプター設定 を参照してください。
- 必要なアドレスがブローカーに設定されている。アドレス、キュー、およびトピック を参照してください。
- ブローカーはクライアントからのアクセスを許可するように、クライアントは必要なクレデンシャルを送信するように設定されます。Broker Security を参照してください。

13.4. AMQ INTERCONNECT への接続

AMQ Interconnect は AMQP 1.0 クライアントであれば機能します。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のアクセスを許可するように、クライアントは必要なクレデンシャルを送信するように設定されます。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 リポジトリー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. 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. **JBOSS INTEGRATION AND AUTOMATION** カテゴリーの **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. パッケージ用のシステムの登録

RPM パッケージを Red Hat Enterprise Linux にインストールするには、システムが登録されている必要があります。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。

詳細は、How to Register and Subscribe a System to the Red Hat Customer Portal を参照してください。

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。

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。それらをすべてインストールする必要はありません。必要なものだけをインストールできます。

プライマリーパッケージは、通常、追加の修飾子がない最もシンプルな名前です。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イスを提供します。

-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。このパッケージに依存するプログラムを構築する際の、コンパイル時に必要になります。

-docs で終わる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とサンプルプログラムが含まれます。

RPM パッケージ を使用する方法は、以下のリソースのいずれかを参照してください。

- Red Hat Enterprise Linux 6 - ソフトウェアのインストールおよび管理
- 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 -ql** コマンドを使用します。

```
$ rpm -ql <package>
```

付録C 例で AMQ ブローカーの使用

AMQ C++ の例では、名前が **examples** というキューが含まれる実行中のメッセージブローカーが必要です。以下の手順に従って、ブローカーをインストールして起動し、キューを定義します。

C.1. ブローカーのインストール

Getting Started with AMQ Brokerの手順に従って、ブローカーをインストール して、ブローカーインスタンスを作成 します。匿名アクセスを有効にします。

以下の手順では、ブローカーインスタンスの場所を **<broker-instance-dir>** と呼びます。

C.2. ブローカーの起動

手順

1. **artemis run** コマンドを使用してブローカーを起動します。

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

2. 起動時にログに記録された重大なエラーがないか、コンソールの出力を確認してください。ブローカーでは、準備が整うと **Server is now live** とログが記録されます。

```
$ 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. キューの作成

新しいターミナルで、**artemis queue** コマンドを使用して **examples** という名前のキューを作成します。

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

プロンプトで質問に Yes または No で回答するように求められます。すべての質問に **N** (いいえ) と回答します。

キューが作成されると、ブローカーはサンプルプログラムで使用できるようになります。

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：2023-01-28 12:22:44 +1000