Red Hat AMQ 2021.Q3

AMQ C++ Client の使用

AMQ Clients 2.10 向け

Last Updated: 2023-02-23

Red Hat AMQ 2021.Q3 AMQ C++ Client 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_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 5
- 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 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

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 14

第5章 API の使用 **15**
- 5.1. メッセージングイベントの処理 15
- 5.2. コンテナーの作成 15
- 5.3. コンテナーアイデンティティーの設定 15

第6章 ネットワーク接続 **17**
- 6.1. 接続 URL 17
- 6.2. 外向き接続の作成 17
- 6.3. 再接続の設定 17
- 6.4. フェイルオーバーの設定 18
- 6.5. 受信接続の許可 19

第7章 セキュリティー **20**
- 7.1. SSL/TLS を使用した接続のセキュリティー保護 20
- 7.2. ユーザーとパスワードを使用した接続 20
- 7.3. SASL 認証の設定 20
- 7.4. KERBEROS を使用した認証 21

第8章 送信者およびレシーバー **22**
- 8.1. オンデマンドでキューとトピックの作成 22
- 8.2. 永続サブスクリプションの作成 23
- 8.3. 共有サブスクリプションの作成 23

第9章 メッセージ配信 **25**
- 9.1. メッセージの送信 25
- 9.2. 送信されたメッセージの追跡 25
- 9.3. メッセージの受信 25

9.4. 受信したメッセージの承認 26
第10章 エラー処理 27
10.1. 例外のキャッチ 27
10.2. 接続およびプロトコルエラーの処理 27
第11章 ロギング 29
11.1. プロトコルロギングの有効化 29
第12章 スレッドおよびスケジューリング 30
12.1. スレッドモデル 30
12.2. スレッドセーフルール 30
12.3. ワークキュー 30
12.4. ウェイクプリミティブ 30
12.5. スケジューリングが遅延しているワーク 31
12.6. 古いバージョンの C++ の使用 31
第13章 ファイルベースの設定 33
13.1. ファイルの場所 33
13.2. ファイルフォーマット 33
13.3. SSL 設定オプション 34
第14章 相互運用性 36
14.1. 他の AMQP クライアントとの相互運用 36
14.2. AMQ JMS での相互運用 40
JMS メッセージタイプ 40
14.3. AMQ BROKER への接続 40
14.4. AMQ INTERCONNECT への接続 41
付録A サブスクリプションの使用 42
A.1. アカウントへのアクセス 42
A.2. サブスクリプションのアクティベート 42
A.3. リリースファイルのダウンロード 42
A.4. パッケージを受信するためのシステムの登録 42
付録B RED HAT ENTERPRISE LINUX パッケージの使用 44
B.1. 概要 44
B.2. パッケージの検索 44
B.3. パッケージのインストール 44
B.4. パッケージ情報のクエリー 44
付録C サンプルでの AMQ BROKER の使用 46
C.1. ブローカーのインストール 46
C.2. ブローカーの起動 46
C.3. キューの作成 46
C.4. ブローカーの停止 47

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 プロパティー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 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。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、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 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 2.10 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

1.2. サポート対象の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) 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++ 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 エンティティーを紹介し、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

エンティティー	説明
コンテナー	接続の最上位のコンテナー。
接続	ネットワーク上の 2 つのピア間の通信用のチャネル。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情報。
配信	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、**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、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 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。
- 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。
- 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++**、**cmake**、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ビルドするには、Visual Studio 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ー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 のバリアントの値 (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、**qpid-proton-cpp-devel** 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. 「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** カテゴリーで **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. **AMQ Clients 2.10.0 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-2.10.0-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、本書全体で **<install-dir>** と呼ばれます。

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

- 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。

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
```

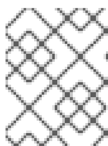
4. **helloworld** プログラムを実行します。

```
$ ./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。

その他の例は、AMQ C++ サンプルのスイート と Qpid Proton C++ の例 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

```
#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。詳細は、3章スタートガイド を参照してください。

```
$ 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

```
#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。詳細は、3章スタートガイド を参照してください。

```
$ 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++ sample 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。これらのメソッドは、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. コンテナーの作成

コンテナーはトップレベルの 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++ が接続を確立すると、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 **proton::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 を設定しないと、コンテナーが構成されると、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 **messaging_handler::on_container_start()** メソッド内で行われます。

例: 送信接続の作成

```
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 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);
```

reconnect を有効にすると、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**、**delay_multiplier**、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されます。

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

```
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。一覧に使い切られると、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. 受信接続の許可

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、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 **proton::container::listen()** メソッドを使用します。

例: 受信接続の許可

```
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 インターフェースでリッスンします。すべての IPv6 インターフェー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

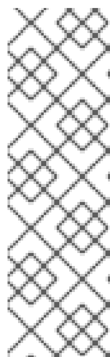
```
proton::connection_options opts {};

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 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** コネクション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

```
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**、および **EXTERNAL** が含まれます。

AMQ C++ 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

```
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. 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。

   ```
   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ネルを表します。送信者と受信側は一方向で、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

```
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： 要求時に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、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。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 receiver を作成し、 **durability_mode** および **expiry_policy** オプションを設定して、持続性を確保するためにレシーバー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.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、durable-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。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、**shared** 機能を設定して共有用の受信側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** イベントは、**proton::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

```
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と、送信側は転送を表す **tracker** オブジェクトへの参照を維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け入れまたは拒否します。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** 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

明示的に配信を許可または拒否するには、**on_message** イベントハンドラーで **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 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傍受するためのイベント処理の関数の上書き

例外の取得は最も基本的なものですが、エラーを処理するための粒度が細かくなります。ハンドラー関数のオーバーライドを使用してエラーを処理しない場合は、例外が発生します。

10.1. 例外のキャッチ

イベント処理関数のオーバーライドを使用してエラーを処理しない場合、コンテナーの **run** メソッドによって例外が発生します。

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** クラスから継承されます。

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** が継承されてから **std::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

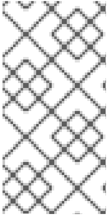
以下の **messaging_handler** メソッドを上書きすると、プロトコルレベルのエラーを処理することができます。

- **on_transport_error(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。

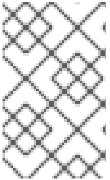
特定のエラーハンドラーのいずれかが上書きされない場合は、デフォルトのエラーハンドラーが呼び出されます。

- **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 スレッドおよびスケジューリング

AMQ C++ は、C++11 以降における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。

12.1. 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)。ただし、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。

handler 接続オプションを使用して、**container::connect()** または **listen_handler::on_accept()** の接続にハンドラー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。

12.2. 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。

12.3. ワークキュー

work_queue インターフェースは、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、**work_queue** が関連付けられています。
- 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** 項目は **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。

12.4. ウェイクプリミティブ

connection::wake() メソッドにより、スレッドは **on_connection_wake()** コールバックをトリガーして、接続でアクティビティーを求めます。これは、**connection** にあるスレッドセーフメソッドのみです。

wake() スレッド間をシグナル化するための軽量の低レベルプリミティブです。

- **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()** 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で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、それについての情報が何であるかを決定するために、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使いやすくなりますが、独自の外部スレッドセーフキューがあり、データをチェックするために接続をウェイクする効率的な方法が必要な場合は、**wake()** が役に立つことがあります。

12.5. スケジューリング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し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。

一定の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、送信者の作業キューで **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。

12.6. 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。スレッド数には 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()** を呼び出す単一の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になります。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。例外は **connection::wake()** のみです。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13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。

13.1. ファイルの場所

設定された場合、AMQ C++ 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 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. **$PWD/connect.json**。ここでの **$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す。

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.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

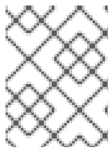
第14章 相互運用性

本章では、AMQ 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。

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 C++ 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 エンコードおよびデコードの後における AMQ C++ タイプ

AMQP 型	エンコード前の AMQ C++ 型	デコード後の AMQ C++ 型
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++ 型	デコード後の AMQ C++ 型
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14.3 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14.4 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 のタイプ	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 のタイプ	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**

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 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。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 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**

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に設定されます。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、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** カテゴリーで **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。必要なものだけ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 -ql** コマンドを使用します。

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

付録C サンプルでの AMQ BROKER の使用

AMQ C++ のサンプルでは、**examples** という名前の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

『**AMQ Broker の使用**』の説明に従い ブロッ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、**<broker-instance-dir>** としてブローカーインスタンスの場所を参照します。

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. コンソールの出力で、起動時にログに記録される重要なエラーの有無を確認します。ブローカーは、準備が整う際に **Server is now live** をログに記録します。

```
$ 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
```

yes または no の質問への回答を求めるプロンプトが表示されます。そのすべてに no (**N**) 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と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: 2021-08-29 15:55:46 +1000