Red Hat AMQ 7.2

Using the AMQ C++ Client

For Use with AMQ Clients 2.1

Last Updated: 2018-07-27

Red Hat AMQ 7.2 Using the AMQ C++ Client

For Use with AMQ Clients 2.1

Legal Notice

Abstract

This guide describes how to install and configure the client, run hands-on examples, and use your client with other AMQ components.

Table of Contents

CHAPTER 1. OVERVIEW

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

AMQ C++ is part of AMQ Clients, a suite of messaging libraries supporting multiple languages and platforms. For an overview of the clients, see AMQ Clients Overview. For information about this release, see AMQ Clients 2.1 Release Notes.

AMQ C++ is based on the Proton API from Apache Qpid.

1.1. KEY FEATURES

- An event-driven API that simplifies integration with existing applications
- SSL/TLS for secure communication
- Flexible SASL authentication
- Automatic reconnect and failover
- Seamless conversion between AMQP and language-native data types
- Access to all the features and capabilities of AMQP 1.0

1.2. SUPPORTED STANDARDS AND PROTOCOLS

AMQ C++ supports the following industry-recognized standards and network protocols:

- Version 1.0 of the Advanced Message Queueing Protocol (AMQP)
- Versions 1.0, 1.1, and 1.2 of the Transport Layer Security (TLS) protocol, the successor to SSL
- Simple Authentication and Security Layer (SASL) mechanisms supported by Cyrus SASL, including ANONYMOUS, PLAIN, SCRAM, EXTERNAL, and GSSAPI (Kerberos)
- Modern TCP with IPv6

1.3. SUPPORTED CONFIGURATIONS

AMQ C++ supports the following OS and language versions:

- Red Hat Enterprise Linux 6 and 7 with GNU C++, compiling as C++03 or C++11
- Microsoft Windows Server 2012 R2 with Microsoft Visual Studio 2013

For more information, see Red Hat AMQ 7 Supported Configurations.

1.4. TERMS AND CONCEPTS

This section introduces the core API entities and describes how they operate together.

Table 1.1. API terms

Entity	Description
Container	A top-level container of connections
Connection	A channel for communication between two peers on a network
Session	A context for sending and receiving messages
Sender	A channel for sending messages to a target
Receiver	A channel for receiving messages from a source
Source	A named point of origin for messages
Target	A named destination for messages
Message	A mutable holder of application data
Delivery	A message transfer

AMQ C++ sends and receives *messages*. Messages are transferred between connected peers over *senders* and *receivers*. Senders and receivers are established over *sessions*. Sessions are established over *connections*. Connections are established between two uniquely identified *containers*. Though a connection can have multiple sessions, often this is not needed. The API allows you to ignore sessions unless you require them.

A sending peer creates a sender to send messages. The sender has a *target* that identifies a queue or topic at the remote peer. A receiving peer creates a receiver to receive messages. The receiver has a *source* that identifies a queue or topic at the remote peer.

The sending of a message is called a *delivery*. The message is the content sent, including all metadata such as headers and annotations. The delivery is the protocol exchange associated with the transfer of that content.

To indicate that a delivery is complete, either the sender or the receiver settles it. When the other side learns that it has been settled, it will no longer communicate about that delivery. The receiver can also indicate whether it accepts or rejects the message.

1.5. DOCUMENT CONVENTIONS

This document uses the following conventions for the **sudo** command and file paths.

The sudo command

In this document, **sudo** is used for any command that requires root privileges. You should always exercise caution when using **sudo**, as any changes can affect the entire system.

For more information about using **sudo**, see The **sudo** Command.

About the use of file paths in this document

In this document, all file paths are valid for Linux, UNIX, and similar operating systems (for example, **`/home/...`**). If you are using Microsoft Windows, you should use the equivalent Microsoft Windows paths (for example, **`C:\Users\...`**).

CHAPTER 2. INSTALLATION

This chapter guides you through the steps to install AMQ C++ in your environment.

2.1. PREREQUISITES

To begin installation, use your subscription to access AMQ distribution files and repositories.

2.2. INSTALLING ON RED HAT ENTERPRISE LINUX

AMQ C++ is distributed as a set of RPM packages for Red Hat Enterprise Linux. Follow these steps to install them.

1. Use the **`subscription-manager`** command to subscribe to the required package repositories.

   Red Hat Enterprise Linux 6

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

   Red Hat Enterprise Linux 7

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

2. Use the **`yum`** command to install the **`qpid-proton-cpp-devel`** and **`qpid-proton-cpp-docs`** packages.

   ```
   $ sudo yum install qpid-proton-cpp-devel qpid-proton-cpp-docs
   ```

In order to compile programs using the API, you will also need to install **`gcc-c++`**, **`cmake`**, and **`make`**.

```
$ sudo yum install gcc-c++ cmake make
```

2.3. INSTALLING ON MICROSOFT WINDOWS

AMQ C++ is distributed as an SDK zip archive for use with Visual Studio. Follow these steps to install it.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at access.redhat.com/downloads.
2. Locate the **Red Hat AMQ Clients** entry in the **JBOSS INTEGRATION AND AUTOMATION** category.
3. Click **Red Hat AMQ Clients**. The **Software Downloads** page opens.
4. Download the **AMQ C++ Client Windows SDK** zip file.
5. Extract the file contents into a directory of your choosing by right-clicking on the zip file and selecting **Extract All**.

CHAPTER 3. GETTING STARTED

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

3.1. PREPARING THE BROKER

The example programs require a running broker with a queue named **examples**. Follow these steps to define the queue and start the broker:

Procedure

1. Install the broker.
2. Create a broker instance. Enable anonymous access.
3. Start the broker instance and check the console for any critical errors logged during startup.

```
$ <broker-instance-dir>/bin/artemis run
...
14:43:20,158 INFO
[org.apache.activemq.artemis.integration.bootstrap] AMQ101000:
Starting ActiveMQ Artemis Server
...
15:01:39,686 INFO  [org.apache.activemq.artemis.core.server]
AMQ221020: Started Acceptor at 0.0.0.0:5672 for protocols [AMQP]
...
15:01:39,691 INFO  [org.apache.activemq.artemis.core.server]
AMQ221007: Server is now live
```

4. Use the **artemis queue** command to create a queue called **examples**.

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

You are prompted to answer a series of questions. For yes or no questions, type **N**. Otherwise, press Enter to accept the default value.

3.2. BUILDING THE EXAMPLES

This section illustrates how to compile the example programs that come with the client API.

1. Create a directory to hold the programs. This example names it "AMQ7C++SmokeTest", but you can use any name you like.

```
$ mkdir AMQ7C++SmokeTest
```

2. Enter the new directory.

```
$ cd AMQ7C++SmokeTest
```

3. Copy all the examples to this directory.

```
$ cp -r /usr/share/proton-0.24.0/examples/cpp .
```

–

NOTE

The example directory name depends on the version of proton we just installed - 0.24.0 is the version as of the writing of this documentation. If a different version is actually installed, this directory name needs to be changed to reflect the actual name installed on the system.

This example compiles all of the examples and then runs the two of interest. They can be compiled like this:

```
$ cmake .
$ make
```

NOTE

It is not recommended to use cmake in the same directory as the source being built. This example does so for simplicity.

Consider creating a directory for builds and run cmake there.

3.3. SENDING AND RECEIVING MESSAGES

The compiled example programs use the broker we started earlier to queue the messages between sending and receiving.

Sending messages

- Use one of the example programs to send 10 messages to a queue named **`examples`**.

```
$ ./simple_send -m 10
```

The command line option **`-m 10`** tells the program to send 10 messages.

This outputs:

```
all messages confirmed
$
```

By default the **`simple_send`** example connects to an AMQP listener on the same machine (IP address **`127.0.0.1`**, port **`5672`**) and sends messages to the AMQP address **`examples`**. This corresponds to the **`examples`** queue that we have configured in the AMQ Broker.

Receiving messages

- Execute the following commands as you did in the previous example.

```
$ ./simple_recv -m 10
```

In this case the command line option **`-m 10`** tells the program to exit after receiving 10 messages.

```
simple_recv listening on 127.0.0.1:5672/examples
```

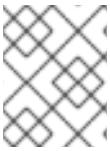
```
{"sequence"=1}
{"sequence"=2}
{"sequence"=3}
{"sequence"=4}
{"sequence"=5}
{"sequence"=6}
{"sequence"=7}
{"sequence"=8}
{"sequence"=9}
{"sequence"=10}
$
```

The **simple_recv** example is similar to the **simple_send** example. It also connects to an AMQP listener on the same machine. By default it subscribes to the AMQP address **examples** and receives 100 messages.

CHAPTER 4. EXAMPLES

This chapter demonstrates the use of AMQ C++ through example programs.

See the Qpid Proton C++ examples for more sample programs.



NOTE

The code presented in this guide uses C++11 features. AMQ C++ is also compatible with C++03, but the code will require minor modifications.

4.1. SENDING MESSAGES

This client program connects to a server using **<connection-url>**, creates a sender for target **<address>**, sends a message containing **<message-body>**, closes the connection, and exits.

Example: Sending messages

```
#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_);
    }

    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;
}
```

Running the example

To run the example program, copy it to a local file, compile it, and execute it from the command line.

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

4.2. RECEIVING MESSAGES

This client program connects to a server using **<connection-url>**, creates a receiver for source **<address>**, and receives messages until it is terminated or it reaches **<count>** messages.

Example: Receiving messages

```
#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_);
    }

    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;
}
```

Running the example

To run the example program, copy it to a local file, compile it, and execute it from the command line.

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

CHAPTER 5. USING THE API

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

5.1. BASIC OPERATION

5.1.1. Handling messaging events

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

Example: Handling messaging events

```
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";
    }
};
```

These are only a few common-case events. The full set is documented in the API reference.

5.1.2. Creating a container

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

Example: Creating a container

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

Setting the container identity
Each container instance has a unique identity called the container ID. When AMQ C++ makes a connection, it sends the container ID to the remote peer. To set the container ID, pass it to the **`proton::container`** constructor.

Example: Setting the container identity

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

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

5.2. NETWORK CONNECTIONS

5.2.1. Connection URLs

Connection URLs encode the information used to establish new connections.

Connection URL syntax

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

- *Scheme* - The connection transport, either **amqp** for unencrypted TCP or **amqps** for TCP with SSL/TLS encryption.
- *Host* - The remote network host. The value can be a hostname or a numeric IP address. IPv6 addresses must be enclosed in square brackets.
- *Port* - The remote network port. This value is optional. The default value is 5672 for the **amqp** scheme and 5671 for the **amqps** scheme.

Connection URL examples

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

5.2.2. Creating outgoing connections

To connect to a remote server, call the **container::connect()** method with a connection URL. This is typically done inside the **messaging_handler::on_container_start()** method.

Example: Creating outgoing connections

```
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 to is open\n";
    }
};
```

See the Section 5.5, “Security” section for information about creating secure connections.

5.2.3. Configuring reconnect

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

AMQ C++ disables reconnect by default. To enable it, set the **reconnect** connection option to an instance of the **reconnect_options** class.

Example: Enabling reconnect

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

opts.reconnect(ropts);

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

With reconnect enabled, if a connection is lost or a connection attempt fails, the client will try again after a brief delay. The delay increases exponentially for each new attempt.

To control the delays between connection attempts, set the **delay**, **delay_multiplier**, and **max_delay** options. All durations are specified in milliseconds.

To limit the number of reconnect attempts, set the **max_attempts** option. Setting it to 0 removes any limit.

Example: Configuring reconnect

```
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);
```

5.2.4. Configuring failover

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

To specify alternate connection endpoints, set the **failover_urls** reconnect option to a list of connection URLs.

Example: Configuring failover

```
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);
```

5.3. MESSAGE DELIVERY

5.3.1. Sending messages

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

Example: Sending messages

```
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);
    }
};
```

5.3.2. Tracking sent messages

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

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

Example: Tracking sent messages

```
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";
}
```

5.3.3. Receiving messages

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

Example: Receiving messages

```
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";
    }
};
```

5.3.4. Acknowledging received messages

To explicitly accept or reject a delivery, use the **delivery::accept()** or **delivery::reject()** methods in the **on_message** event handler.

Example: Acknowledging received messages

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

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

5.4. ERROR HANDLING

Errors in AMQ C++ can be handled in two different ways.

- Catching exceptions
- Overriding virtual functions to handle AMQP protocol or connection errors

Catching exceptions
Catching exceptions is the most basic, but least granular, way to handle errors. If an error is not handled using an override in a handler routine, an exception will be thrown and can be caught and handled. An exception thrown in this way will be thrown by the container's **run** method.

All of the exceptions that can be thrown by AMQ C++ are descended from **proton::error**, which in turn is a subclass of **std::runtime_error** (which is a subclass of **std::exception**).

The code example below illustrates how a block could be written to catch any exception thrown from AMQ C++.

Example: API-Specific exception handling

```
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
}
```

If you require no API-specific exception handling, you only need to catch **`std::exception`** since **`proton::error`** descends from it.

Example: General exception handling

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

NOTE

Because all exceptions in a C++ program descend from **`std::exception`**, you can write a code block to wrap your **`main`** method and display information about any **`std::exception`** errors.

Handling connection and protocol errors

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

- **`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&)`**

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

If not overridden the default error handler will be called with an indication of the error condition that occurred.

There is also a default error handler:

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

If one of the more specific error handlers is not overridden, this will be called.

NOTE

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

5.5. SECURITY

5.5.1. Securing connections with SSL/TLS

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

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

Example: Enabling SSL/TLS

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

5.5.2. Connecting with a user and password

AMQ C++ can authenticate connections with a user and password.

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

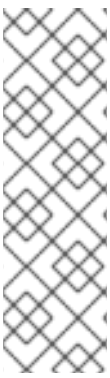
Example: Connecting with a user and password

```
proton::connection_options opts {};
opts.user("alice");
opts.password("secret");

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

5.5.3. Configuring SASL authentication

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



NOTE

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

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

By default, AMQ C++ allows all of the mechanisms supported by the local SASL library configuration. To

restrict the allowed mechanisms and thereby control what mechanisms can be negotiated, use the **sasl_allowed_mechs** connection option. It takes a string containing a space-separated list of mechanism names.

Example: Configuring SASL authentication

```
proton::connection_options opts {};
opts.sasl_allowed_mechs("ANONYMOUS");

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

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

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

Example: Disabling SASL

```
proton::connection_options opts {};
opts.sasl_enabled(false);

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

5.5.4. Authenticating using Kerberos

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

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

   ```
   proton::connection_options opts {};
   opts.sasl_allowed_mechs("GSSAPI");

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

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

   ```
   $ kinit USER@REALM
   ```

4. Run the client program.

5.6. TIMERS

AMQ C++ has the ability to execute code after a delay. You can use this to implement time-based behaviors in your application, such as periodically scheduled work or timeouts.

5.6.1. Scheduling deferred work

To defer work for a fixed amount of time, use the **`schedule`** method to set the delay and register a function defining the work.

Example: Sending a message after a delay

```
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);
    }
}
```

This example uses the **`schedule`** method on the work queue of the sender in order to establish it as the execution context for the work.

5.7. MORE INFORMATION

For more information, see the API reference.

CHAPTER 6. MULTITHREADING

AMQ C++ supports full multithreading with C++11 and later. Limited multithreading is possible with older versions of C++. See Section 6.5, “Using older versions of C++”.

6.1. THREADING MODEL

The **`container`** object can handle multiple connections concurrently. As AMQP events occur on connections, the container calls **`messaging_handler`** callback functions. Callbacks for any one connection are serialized (not called concurrently), but callbacks for different connections can be safely executed in parallel.

You can assign a handler to a connection in **`container::connect()`** or **`listen_handler::on_accept()`** using the **`handler`** connection option. We recommend creating a separate handler for each connection. That way the handler does not need locks or other synchronization to protect it against concurrent use by library threads. If any non-library threads use the handler concurrently, then you will need synchronization.

6.2. THREAD-SAFETY RULES

The **`connection`**, **`session`**, **`sender`**, **`receiver`**, **`tracker`**, and **`delivery`** objects are not thread-safe and are subject to the following rules.

1. You must use them only from a **`messaging_handler`** callback or a **`work_queue`** function.
2. You must not use objects belonging to one connection from a callback for another connection.
3. You can store AMQ C++ objects in member variables for use in a later callback, provided you respect rule two.

The **`message`** object is a value type with the same threading constraints as a standard C++ built-in type. It cannot be concurrently modified.

6.3. WORK QUEUES

The **`work_queue`** interface provides a safe way to communicate between different connection handlers or between non-library threads and connection handlers.

- Each connection has an associated **`work_queue`**.
- The work queue is thread-safe (C++11 or greater). Any thread can add work.
- A **`work`** item is a **`std::function`**, and bound arguments are called like an event callback.

When the library calls the work function, it will be serialized safely so that you can treat the work function like an event callback and safely access the handler and AMQ C++ objects stored on it.

6.4. THE WAKE PRIMITIVE

The **`connection::wake()`** method allows any thread to prompt activity on a connection by triggering an **`on_connection_wake()`** callback. This is the only thread-safe method on **`connection`**.

`wake()` is a lightweight, low-level primitive for signaling between threads.

- It does not carry any code or data, unlike **`work_queue`**.
- Multiple calls to **`wake()`** might be coalesced into a single **`on_connection_wake()`**.
- Calls to **`on_connection_wake()`** can occur without any application call to **`wake()`** since the library uses **`wake()`** internally.

The semantics of **`wake()`** are similar to **`std::condition_variable::notify_one()`**. There will be a wakeup, but there must be some shared application state to determine why the wakeup occurred and what, if anything, to do about it.

Work queues are easier to use in many instances, but **`wake()`** may be useful if you already have your own external thread-safe queues and need an efficient way to wake a connection to check them for data.

6.5. USING OLDER VERSIONS OF C++

Before C++11 there was no standard support for threading in C++. You can use AMQ C++ with threads but with the following limitations.

- The container will not create threads. It will only use the single thread that calls **`container::run()`**.
- None of the AMQ C++ library classes are thread-safe, including **`container`** and **`work_queue`**. You need an external lock to use **`container`** in multiple threads. The only exception is **`connection::wake()`**. It is thread-safe even in older C++.

CHAPTER 7. INTEROPERABILITY

This chapter discusses how to use AMQ C++ in combination with other AMQ components. For an overview of the compatibility of AMQ components, see the product introduction.

7.1. INTEROPERATING WITH OTHER AMQP CLIENTS

AMQP messages are composed using the AMQP type system. This common format is one of the reasons AMQP clients in different languages are able to interoperate with each other.

When sending messages, AMQ C++ automatically converts language-native types to AMQP-encoded data. When receiving messages, the reverse conversion takes place.

NOTE

More information about AMQP types is available at the interactive type reference maintained by the Apache Qpid project.

Table 7.1. AMQP types

AMQP type	Description
`null`	An empty value
`boolean`	A true or false value
`char`	A single Unicode character
`string`	A sequence of Unicode characters
`binary`	A sequence of bytes
`byte`	A signed 8-bit integer
`short`	A signed 16-bit integer
`int`	A signed 32-bit integer
`long`	A signed 64-bit integer
`ubyte`	An unsigned 8-bit integer
`ushort`	An unsigned 16-bit integer
`uint`	An unsigned 32-bit integer
`ulong`	An unsigned 64-bit integer
`float`	A 32-bit floating point number

AMQP type	Description
`double`	A 64-bit floating point number
`array`	A sequence of values of a single type
`list`	A sequence of values of variable type
`map`	A mapping from distinct keys to values
`uuid`	A universally unique identifier
`symbol`	A 7-bit ASCII string from a constrained domain
`timestamp`	An absolute point in time

Table 7.2. AMQ C++ types before encoding and after decoding

AMQP type	AMQ C++ type before encoding	AMQ C++ type after decoding
`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 type	AMQ C++ type before encoding	AMQ C++ type after decoding
`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`

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

AMQ C++ type before encoding	AMQ JavaScript type	AMQ .NET type
`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`

AMQ C++ type before encoding	AMQ JavaScript type	AMQ .NET type
`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`

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

AMQ C++ type before encoding	AMQ Python type	AMQ Ruby type
`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`

AMQ C++ type before encoding	AMQ Python type	AMQ Ruby type
`float`	`float`	`Float`
`double`	`float`	`Float`
`std::vector`	`list`	`Array`
`std::map`	`dict`	`Hash`
`proton::uuid`	-	-
`proton::symbol`	`str`	`Symbol`
`proton::timestamp`	`long`	`Time`

7.2. INTEROPERATING WITH AMQ JMS

AMQP defines a standard mapping to the JMS messaging model. This section discusses the various aspects of that mapping. For more information, see the AMQ JMS Interoperability chapter.

JMS message types

AMQ C++ provides a single message type whose body type can vary. By contrast, the JMS API uses different message types to represent different kinds of data. The table below indicates how particular body types map to JMS message types.

For more explicit control of the resulting JMS message type, you can set the `x-opt-jms-msg-type` message annotation. See the AMQ JMS Interoperability chapter for more information.

Table 7.5. AMQ C++ and JMS message types

AMQ C++ body type	JMS message type
`std::string`	`TextMessage`
`nullptr`	`TextMessage`
`proton::binary`	`BytesMessage`
Any other type	`ObjectMessage`

7.3. CONNECTING TO AMQ BROKER

AMQ Broker is designed to interoperate with AMQP 1.0 clients. Check the following to ensure the broker is configured for AMQP messaging.

- Port 5672 in the network firewall is open.
- The AMQ Broker AMQP acceptor is enabled. See Configuring Network Access.

- The necessary addresses are configured on the broker. See Addresses, Queues, and Topics.
- The broker is configured to permit access from your client, and the client is configured to send the required credentials. See Broker Security.

7.4. CONNECTING TO AMQ INTERCONNECT

AMQ Interconnect works with any AMQP 1.0 client. Check the following to ensure the components are configured correctly.

- Port 5672 in the network firewall is open.
- The router is configured to permit access from your client, and the client is configured to send the required credentials. See Interconnect Security.

APPENDIX A. USING YOUR SUBSCRIPTION

AMQ is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

Accessing your account

1. Go to access.redhat.com.
2. If you do not already have an account, create one.
3. Log in to your account.

Activating a subscription

1. Go to access.redhat.com.
2. Navigate to **My Subscriptions**.
3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

Downloading zip and tar files

To access zip or tar files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at access.redhat.com/downloads.
2. Locate the **Red Hat AMQ** entries in the **JBOSS INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

Registering your system for packages

To install RPM packages on Red Hat Enterprise Linux, your system must be registered. If you are using zip or tar files, this step is not required.

1. Go to access.redhat.com.
2. Navigate to **Registration Assistant**.
3. Select your OS version and continue to the next page.
4. Use the listed command in your system terminal to complete the registration.

To learn more see How to Register and Subscribe a System to the Red Hat Customer Portal.

Revised on 2018-07-27 17:59:16 EDT