



## Red Hat AMQ 7.5

### Using the AMQ .NET Client

For Use with AMQ Clients 2.6



# Red Hat AMQ 7.5 Using the AMQ .NET Client

---

For Use with AMQ Clients 2.6

## Legal Notice

Copyright © 2020 Red Hat, Inc.

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<sup>®</sup> is the registered trademark of Linus Torvalds in the United States and other countries.

Java<sup>®</sup> is a registered trademark of Oracle and/or its affiliates.

XFS<sup>®</sup> is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL<sup>®</sup> is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js<sup>®</sup> 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<sup>®</sup> 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.

## 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

<b>CHAPTER 1. OVERVIEW</b> .....	<b>4</b>
1.1. KEY FEATURES	4
1.2. SUPPORTED STANDARDS AND PROTOCOLS	4
1.3. SUPPORTED CONFIGURATIONS	4
1.4. TERMS AND CONCEPTS	5
<b>CHAPTER 2. INSTALLATION</b> .....	<b>6</b>
2.1. PREREQUISITES	6
2.2. INSTALLING ON RED HAT ENTERPRISE LINUX	6
2.3. INSTALLING ON MICROSOFT WINDOWS	6
<b>CHAPTER 3. GETTING STARTED</b> .....	<b>8</b>
3.1. PREPARING THE BROKER	8
3.2. BUILDING THE EXAMPLES	8
3.3. RUNNING HELLO WORLD	8
<b>CHAPTER 4. EXAMPLES</b> .....	<b>10</b>
4.1. SENDING MESSAGES	10
Running the example	11
4.2. RECEIVING MESSAGES	11
Running the example	12
<b>CHAPTER 5. USING THE API</b> .....	<b>14</b>
5.1. NETWORK CONNECTIONS	14
5.1.1. Creating outgoing connections	14
5.2. SENDERS AND RECEIVERS	14
5.2.1. Creating queues and topics on demand	14
5.2.2. Creating durable subscriptions	15
5.2.3. Creating shared subscriptions	16
5.3. MESSAGE DELIVERY	17
5.3.1. Sending messages	17
5.3.2. Receiving messages	17
5.4. SECURITY	17
5.4.1. Connecting with a user and password	17
5.4.2. Configuring SASL authentication	18
5.4.3. Configuring an SSL/TLS transport	18
5.5. LOGGING	18
5.5.1. Setting the log output level	18
5.5.2. Enabling protocol logging	18
<b>CHAPTER 6. INTEROPERABILITY</b> .....	<b>20</b>
6.1. INTEROPERATING WITH OTHER AMQP CLIENTS	20
6.2. INTEROPERATING WITH AMQ JMS	24
JMS message types	24
6.3. CONNECTING TO AMQ BROKER	24
6.4. CONNECTING TO AMQ INTERCONNECT	25
<b>APPENDIX A. MANAGING CERTIFICATES</b> .....	<b>26</b>
A.1. INSTALLING CERTIFICATE AUTHORITY CERTIFICATES	26
A.2. INSTALLING CLIENT CERTIFICATES	26
A.3. HELLO WORLD USING CLIENT CERTIFICATES	27
<b>APPENDIX B. EXAMPLE PROGRAMS</b> .....	<b>28</b>

---

B.1. PREREQUISITES	28
B.2. HELLOWORLD SIMPLE	28
HelloWorld-simple command line options	28
HelloWorld-simple sample invocation	28
B.3. HELLOWORLD ROBUST	28
HelloWorld-robust command line options	29
HelloWorld-robust sample invocation	29
B.4. INTEROP.DRAIN.CS, INTEROP.SPOUT.CS (PERFORMANCE EXERCISER)	29
Interop.Drain command line options	29
Interop.Spout command line options	30
Interop.Spout and Interop.Drain sample invocation	30
B.5. INTEROP.CLIENT, INTEROP.SERVER (REQUEST-RESPONSE)	30
Interop.Client command line options	31
Interop.Server command line options	31
Interop.Client, Interop.Server sample invocation	31
PeerToPeer.Client command line options	31
PeerToPeer.Server command line options	31
PeerToPeer.Client, PeerToPeer.Server sample invocation	31
<b>APPENDIX C. USING YOUR SUBSCRIPTION</b> .....	<b>33</b>
Accessing your account	33
Activating a subscription	33
Downloading ZIP and TAR files	33
Registering your system for packages	33



# CHAPTER 1. OVERVIEW

AMQ .NET is a lightweight AMQP 1.0 library for the *.NET Framework*. It enables you to write .NET applications that send and receive AMQP messages.

AMQ .NET 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 2.6 Release Notes](#).

AMQ .NET is based on [AMQP.Net Lite](#). For detailed API documentation, see the [AMQ .NET API reference](#).

## 1.1. KEY FEATURES

- SSL/TLS for secure communication
- Flexible SASL authentication
- Seamless conversion between AMQP and native data types
- Access to all the features and capabilities of AMQP 1.0
- An integrated development environment with full *IntelliSense* API documentation

## 1.2. SUPPORTED STANDARDS AND PROTOCOLS

AMQ .NET supports the following industry-recognized standards and network protocols:

- Version 1.0 of the [Advanced Message Queueing Protocol](#) (AMQP)
- Versions 1.1 and 1.2 of the [Transport Layer Security](#) (TLS) protocol, the successor to SSL
- [Simple Authentication and Security Layer](#) (SASL) mechanisms ANONYMOUS, PLAIN, and EXTERNAL
- Modern [TCP](#) with [IPv6](#)

## 1.3. SUPPORTED CONFIGURATIONS

AMQ .NET supports the following OS and language versions:

- Red Hat Enterprise Linux 7 with .NET Core 2.1
- Microsoft Windows 10 Pro with .NET Core 2.1 or .NET Framework 4.5
- Microsoft Windows Server 2012 R2 and 2016 with .NET Core 2.1 or .NET Framework 4.5

AMQ .NET is supported in combination with the following AMQ components and versions:

- All versions of AMQ Broker
- All versions of AMQ Interconnect
- All versions of AMQ Online



- A-MQ 6 versions 6.2.1 and higher

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
<b>Connection</b>	A channel for communication between two peers on a network
<b>Session</b>	A context for sending and receiving messages
<b>Sender link</b>	A channel for sending messages to a target
<b>Receiver link</b>	A channel for receiving messages from a source
<b>Source</b>	A named point of origin for messages
<b>Target</b>	A named destination for messages
<b>Message</b>	A mutable holder of application data

AMQ .NET sends and receives *messages*. Messages are transferred between connected peers over *links*. Links are established over *sessions*. Sessions are established over *connections*.

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

## CHAPTER 2. INSTALLATION

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

### 2.1. PREREQUISITES

To begin installation, [use your subscription](#) to access AMQ distribution files and repositories.

Building applications on Red Hat Enterprise Linux requires the .NET Core 2.1 developer tools. For information about installing the tools, see the [.NET Core 2.1 Getting Started Guide](#).

Building applications on Windows with AMQ .NET requires *Visual Studio 2012* or later. Solution files built by *Visual Studio 2012* and *Visual Studio 2013* are supplied in the kit, and these files can be opened by any later version of *Visual Studio*.

### 2.2. INSTALLING ON RED HAT ENTERPRISE LINUX

Use the following steps to install the .NET Core SDK:

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at [access.redhat.com/downloads](https://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 .NET 2.6.0 .NET Core SDK**.zip file.
5. Use the **unzip** command to extract the file contents into a directory of your choosing:

```
$ unzip amq-dotnet-2.6.0-core-sdk.zip
Archive: amq-dotnet-2.6.0-core-sdk.zip
  creating: amq-dotnet-2.6.0-core-sdk/
  creating: amq-dotnet-2.6.0-core-sdk/bin/
  creating: amq-dotnet-2.6.0-core-sdk/bin/netstandard2.0/
...
```

When you extract the contents of the .zip file, a directory named **amq-dotnet-2.6.0-core-sdk** is created. This is the top-level directory of the SDK and is referred to as **<install-dir>** throughout this document.

### 2.3. INSTALLING ON MICROSOFT WINDOWS

Use the following steps to install the Windows SDK:

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at [access.redhat.com/downloads](https://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 .NET 2.6.0 Windows SDK**.zip file.

5. Extract the file contents into a directory by right-clicking on the .zip file and selecting **Extract All**.

When you extract the contents of the .zip file, a directory named **amq-dotnet-2.6.0-sdk** is created. This is the top-level directory of the SDK and is referred to as **<install-dir>** throughout this document.

## CHAPTER 3. GETTING STARTED

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

### 3.1. PREPARING THE BROKER

The example programs require a running broker with a queue named **service\_queue**. 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 **service\_queue**.

```
<broker-instance-dir>/bin/artemis queue create --name service_queue --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

AMQ .NET provides example solution and project files to help users get started quickly.

Navigate to **<install-dir>** and open one of the solution files.

Solution File	Visual Studio Version
<b>amqp.sln</b>	Visual Studio 2013
<b>amqp-vs2012.sln</b>	Visual Studio 2012

Select **Build Solution** from the **Build** menu to compile the solution.

### 3.3. RUNNING HELLO WORLD

Open a command prompt window and execute these commands to send and receive a message.

```
D:\>cd <install-dir>\bin\Debug
D:\>HelloWorld-Simple
Hello world!
```

## CHAPTER 4. EXAMPLES

This chapter demonstrates the use of AMQ .NET through example programs.

For more examples, see the [AMQ .NET example suite](#).

### 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

```
namespace SimpleSend
{
    using System;
    using Amqp; 1

    class SimpleSend
    {
        static void Main(string[] args)
        {
            string url = (args.Length > 0) ? args[0] : 2
                "amqp://guest:guest@127.0.0.1:5672";
            string target = (args.Length > 1) ? args[1] : "examples"; 3
            int count = (args.Length > 2) ? Convert.ToInt32(args[2]) : 10; 4

            Address peerAddr = new Address(url); 5
            Connection connection = new Connection(peerAddr); 6
            Session session = new Session(connection);
            SenderLink sender = new SenderLink(session, "send-1", target); 7

            for (int i = 0; i < count; i++)
            {
                Message msg = new Message("simple " + i); 8
                sender.Send(msg); 9
                Console.WriteLine("Sent: " + msg.Body.ToString());
            }

            sender.Close(); 10
            session.Close();
            connection.Close();
        }
    }
}
```

**1** **using Amqp;** Imports types defined in the Amqp namespace. Amqp is defined by a project reference to library file *Amqp.Net.dll* and provides all the classes, interfaces, and value types associated with AMQ .NET.

**2** Command line arg[0] **url** is the network address of the host or virtual host for the AMQP connection. This string describes the connection transport, the user and password credentials, and the port number for the connection on the remote host. *url* may address a broker, a standalone

peer, or an ingress point for a router network.

- 3 Command line arg[1] **target** is the name of the message destination endpoint or resource in the remote host.
- 4 Command line arg[2] **count** is the number of messages to send.
- 5 **peerAddr** is a structure required for creating an AMQP connection.
- 6 Create the AMQP connection.
- 7 **sender** is a client *SenderLink* over which messages may be sent. The link is arbitrarily named *send-1*. Use link names that make sense in your environment and will help to identify traffic in a busy system. Link names are not restricted but must be unique within the same session.
- 8 In the message send loop a new message is created.
- 9 The message is sent to the AMQP peer.
- 10 After all messages are sent then the protocol objects are shut down in an orderly fashion.

### Running the example

```
D:\lite_kit\amqpnetlite\bin\Debug>simple_send amqp://10.10.59.182 service_queue
Sent: simple 0
Sent: simple 1
Sent: simple 2
Sent: simple 3
Sent: simple 4
Sent: simple 5
Sent: simple 6
Sent: simple 7
Sent: simple 8
Sent: simple 9

D:\lite_kit\amqpnetlite\bin\Debug>
```

## 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

```
namespace SimpleRecv
{
    using System;
    using Amqp; 1

    class SimpleRecv
    {
        static void Main(string[] args)
        {
            string url = (args.Length > 0) ? args[0] : 2
```

```

    "amqp://guest:guest@127.0.0.1:5672";
    string source = (args.Length > 1) ? args[1] : "examples";
    int count = (args.Length > 2) ? Convert.ToInt32(args[2]) : 10;

    Address peerAddr = new Address(url);
    Connection connection = new Connection(peerAddr);
    Session session = new Session(connection);
    ReceiverLink receiver = new ReceiverLink(session, "rcv-1", source);

    for (int i = 0; i < count; i++)
    {
        Message msg = receiver.Receive();
        receiver.Accept(msg);
        Console.WriteLine("Received: " + msg.Body.ToString());
    }

    receiver.Close();
    session.Close();
    connection.Close();
}
}
}

```

- 1 **using Amqp;** Imports types defined in the Amqp namespace. Amqp is defined by a project reference to library file *Amqp.Net.dll* and provides all the classes, interfaces, and value types associated with AMQ .NET.
- 2 Command line arg[0] **url** is the network address of the host or virtual host for the AMQP connection. This string describes the connection transport, the user and password credentials, and the port number for the connection on the remote host. *url* may address a broker, a standalone peer, or an ingress point for a router network.
- 3 Command line arg[1] **source** is the name of the message source endpoint or resource in the remote host.
- 4 Command line arg[2] **count** is the number of messages to send.
- 5 **peerAddr** is a structure required for creating an AMQP connection.
- 6 Create the AMQP connection.
- 7 **receiver** is a client *ReceiverLink* over which messages may be received. The link is arbitrarily named *rcv-1*. Use link names that make sense in your environment and will help to identify traffic in a busy system. Link names are not restricted but must be unique within the same session.
- 8 A message is received.
- 9 The messages is accepted. This transfers ownership of the message from the peer to the receiver.
- 10 After all messages are received then the protocol objects are shut down in an orderly fashion.

## Running the example

```
D:\lite_kit\amqpnetlite\bin\Debug>simple_rcv amqp://10.10.59.182 service_queue
```



```
Received: simple 0  
Received: simple 1  
Received: simple 2  
Received: simple 3  
Received: simple 4  
Received: simple 5  
Received: simple 6  
Received: simple 7  
Received: simple 8  
Received: simple 9
```

```
D:\lite_kit\amqpnetlite\bin\Debug>
```

## CHAPTER 5. USING THE API

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

For more information, see the [AMQ .NET API reference](#) and [AMQ .NET example suite](#).

### 5.1. NETWORK CONNECTIONS

#### 5.1.1. Creating outgoing connections

This section describes the standard format of the Connection URI string used to connect to an AMQP remote peer.

```
scheme = ( "amqp" | "amqps" )  
host = ( <fully qualified domain name> | <hostname> | <numeric IP address> )
```

```
URI = scheme "://" [user ":" [password] "@"] host [":" port]
```

- **scheme amqp** - connection uses TCP transport and sets the default port to 5672.
- **scheme amqps** - connection uses SSL/TLS transport and sets the default port to 5671.
- **user** - optional connection authentication user name. If the *user* name is present then the client initiates an AMQP SASL user credential exchange during connection startup.
- **password** - optional connection authentication password.
- **host** - network host to which the connection is directed.
- **port** - optional network port to which the connection is directed. The default *port* value is determined by the AMQP transport scheme.

Connection URI Examples

```
amqp://127.0.0.1  
amqp://amqpserver.example.com:5672  
amqps://joe:somepassword@bigbank.com  
amqps://sue:secret@test.example.com:21000
```

### 5.2. SENDERS AND RECEIVERS

The client uses sender and receiver links to represent channels for delivering messages. Senders and receivers are unidirectional, with a source end for the message origin, and a target end for the message destination.

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

#### 5.2.1. Creating queues and topics on demand

Some message servers support on-demand creation of queues and topics. When a sender or receiver is attached, the server uses the sender target address or the receiver source address to create a queue or topic with a name matching the address.

The message server typically defaults to creating either a queue (for one-to-one message delivery) or a topic (for one-to-many message delivery). The client can indicate which it prefers by setting the **queue** or **topic** capability on the source or target.

To select queue or topic semantics, follow these steps:

1. Configure your message server for automatic creation of queues and topics. This is often the default configuration.
2. Set either the **queue** or **topic** capability on your sender target or receiver source, as in the examples below.

### Example: Sending to a queue created on demand

```
Target target = new Target() {
    Address = "jobs",
    Capabilities = new Symbol[] {"queue"},
};

SenderLink sender = new SenderLink(session, "s1", target, null);
```

### Example: Receiving from a topic created on demand

```
Source source = new Source() {
    Address = "notifications",
    Capabilities = new Symbol[] {"topic"},
};

ReceiverLink receiver = new ReceiverLink(session, "r1", source, null);
```

For more information, see the following examples:

- [QueueSend.cs](#)
- [QueueReceive.cs](#)
- [TopicSend.cs](#)
- [TopicReceive.cs](#)

## 5.2.2. Creating durable subscriptions

A durable subscription is a piece of state on the remote server representing a message receiver. Ordinarily, message receivers are discarded when a client closes. However, because durable subscriptions are persistent, clients can detach from them and then re-attach later. Any messages received while detached are available when the client re-attaches.

Durable subscriptions are uniquely identified by combining the client container ID and receiver name to form a subscription ID. These must have stable values so that the subscription can be recovered.

To create a durable subscription, follow these steps:

1. Set the connection container ID to a stable value, such as **client-1**:

```
Connection conn = new Connection(new Address(connUrl),
```

```
SaslProfile.Anonymous,
new Open() { ContainerId = "client-1" },
null);
```

2. Configure the receiver source for durability by setting the **Durable** and **ExpiryPolicy** properties:

```
Source source = new Source()
{
    Address = "notifications",
    Durable = 2,
    ExpiryPolicy = new Symbol("never"),
};
```

3. Create a receiver with a stable name, such as **sub-1**, and apply the source properties:

```
ReceiverLink receiver = new ReceiverLink(session, "sub-1", source, null);
```

To detach from a subscription, close the connection without explicitly closing the receiver. To terminate the subscription, close the receiver directly.

For more information, see the [DurableSubscribe.cs example](#).

### 5.2.3. Creating shared subscriptions

A shared subscription is a piece of state on the remote server representing one or more message receivers. Because it is shared, multiple clients can consume from the same stream of messages.

The client configures a shared subscription by setting the **shared** capability on the receiver source.

Shared subscriptions are uniquely identified by combining the client container ID and receiver name to form a subscription ID. These must have stable values so that multiple client processes can locate the same subscription. If the **global** capability is set in addition to **shared**, the receiver name alone is used to identify the subscription.

To create a shared subscription, follow these steps:

1. Set the connection container ID to a stable value, such as **client-1**:

```
Connection conn = new Connection(new Address(connUrl),
    SaslProfile.Anonymous,
    new Open() { ContainerId = "client-1" },
    null);
```

2. Configure the receiver source for sharing by setting the **shared** capability:

```
Source source = new Source()
{
    Address = "notifications",
    Capabilities = new Symbol[] { "shared" },
};
```

3. Create a receiver with a stable name, such as **sub-1**, and apply the source properties:

```
ReceiverLink receiver = new ReceiverLink(session, "sub-1", source, null);
```

For more information, see the [SharedSubscribe.cs example](#).

## 5.3. MESSAGE DELIVERY

### 5.3.1. Sending messages

To send a message, create a connection, session, and sender link. Then call the **Sender.Send()** method with a **Message** object.

#### Example: Sending messages

```
Connection connection = new Connection(new Address("amqp://example.com"));
Session session = new Session(connection);
SenderLink sender = new SenderLink(session, "sender-1", "jobs");

Message message = new Message("job-content");
sender.Send(message);
```

For more information, see the [Send.cs example](#).

### 5.3.2. Receiving messages

To receive a message, create a connection, session, and receiver link. Then call the **Receiver.Receive()** method and use the returned **Message** object.

#### Example: Receiving messages

```
Connection connection = new Connection(new Address("amqp://example.com"));
Session session = new Session(connection);
ReceiverLink receiver = new ReceiverLink(session, "receiver-1", "jobs");

Message message = receiver.Receive();
receiver.Accept(message);
```

The **Receiver.Accept()** call tells the remote peer that the message was received and processed.

For more information, see the [Receive.cs example](#).

## 5.4. SECURITY

### 5.4.1. Connecting with a user and password

AMQ .NET can authenticate connections with a user and password.

To specify the credentials used for authentication, set the **user** and **password** fields in the connection URL.

#### Example: Connecting with a user and password

```
Address addr = new Address("amqp://<user>:<password>@example.com");  
Connection conn = new Connection(addr);
```

### 5.4.2. Configuring SASL authentication

Client connections to remote peers may exchange SASL user name and password credentials. The presence of the *user* field in the connection URI controls this exchange. If *user* is specified then SASL credentials are exchanged; if *user* is absent then the SASL credentials are not exchanged.

By default the client supports **EXTERNAL**, **PLAIN**, and **ANONYMOUS** SASL mechanisms.

### 5.4.3. Configuring an SSL/TLS transport

Secure communication with servers is achieved using SSL/TLS. A client may be configured for SSL/TLS Handshake only or for SSL/TLS Handshake and client certificate authentication. See the [Managing Certificates](#) section for more information.



#### NOTE

[TLS Server Name Indication](#) (SNI) is handled automatically by the client library. However, SNI is signaled only for addresses that use the *amqps* transport scheme where the host is a fully qualified domain name or a host name. SNI is not signaled when the host is a numeric IP address.

## 5.5. LOGGING

Logging is important in troubleshooting and debugging. By default logging is turned off. To enable logging a user must set a logging level and provide a delegate function to receive the log messages.

### 5.5.1. Setting the log output level

The library emits log traces at different levels:

- Error
- Warning
- Information
- Verbose

The lowest log level, *Error*, will trace only error events and produce the fewest log messages. A higher log level includes all the log levels below it and generates a larger volume of log messages.

```
// Enable Error logs only.  
Trace.TraceLevel = TraceLevel.Error
```

```
// Enable Verbose logs. This includes logs at all log levels.  
Trace.TraceLevel = TraceLevel.Verbose
```

### 5.5.2. Enabling protocol logging

The Log level *Frame* is handled differently. Setting trace level *Frame* enables tracing outputs for AMQP protocol headers and frames.

Tracing at one of the other log levels must be ORed with *Frame* to get normal tracing output and AMQP frame tracing at the same time. For example

```
// Enable just AMQP frame tracing
Trace.TraceLevel = TraceLevel.Frame;
```

```
// Enable AMQP Frame logs, and Warning and Error logs
Trace.TraceLevel = TraceLevel.Frame | TraceLevel.Warning;
```

The following code writes AMQP frames to the console.

### Example: Logging delegate

```
Trace.TraceLevel = TraceLevel.Frame;
Trace.TraceListener = (f, a) => Console.WriteLine(
    DateTime.Now.ToString("[hh:mm:ss.fff]") + " " + string.Format(f, a));
```

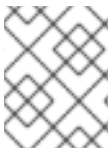
## CHAPTER 6. INTEROPERABILITY

This chapter discusses how to use AMQ .NET in combination with other AMQ components. For an overview of the compatibility of AMQ components, see the [product introduction](#).

### 6.1. 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 .NET 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 6.1. AMQP types

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



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

Table 6.2. AMQ .NET types before encoding and after decoding

AMQP type	AMQ .NET type before encoding	AMQ .NET type after decoding
<b>null</b>	<b>null</b>	<b>null</b>
<b>boolean</b>	<b>System.Boolean</b>	<b>System.Boolean</b>
<b>char</b>	<b>System.Char</b>	<b>System.Char</b>
<b>string</b>	<b>System.String</b>	<b>System.String</b>
<b>binary</b>	<b>System.Byte[]</b>	<b>System.Byte[]</b>
<b>byte</b>	<b>System.SByte</b>	<b>System.SByte</b>
<b>short</b>	<b>System.Int16</b>	<b>System.Int16</b>
<b>int</b>	<b>System.Int32</b>	<b>System.Int32</b>
<b>long</b>	<b>System.Int64</b>	<b>System.Int64</b>
<b>ubyte</b>	<b>System.Byte</b>	<b>System.Byte</b>
<b>ushort</b>	<b>System.UInt16</b>	<b>System.UInt16</b>
<b>uint</b>	<b>System.UInt32</b>	<b>System.UInt32</b>
<b>ulong</b>	<b>System.UInt64</b>	<b>System.UInt64</b>

AMQP type	AMQ .NET type before encoding	AMQ .NET type after decoding
float	<b>System.Single</b>	<b>System.Single</b>
double	<b>System.Double</b>	<b>System.Double</b>
list	<b>Amqp.List</b>	<b>Amqp.List</b>
map	<b>Amqp.Map</b>	<b>Amqp.Map</b>
uuid	<b>System.Guid</b>	<b>System.Guid</b>
symbol	<b>Amqp.Symbol</b>	<b>Amqp.Symbol</b>
timestamp	<b>System.DateTime</b>	<b>System.DateTime</b>

Table 6.3. AMQ .NET and other AMQ client types (1 of 2)

AMQ .NET type before encoding	AMQ C++ type	AMQ JavaScript type
null	<b>nullptr</b>	<b>null</b>
<b>System.Boolean</b>	<b>bool</b>	<b>boolean</b>
<b>System.Char</b>	<b>wchar_t</b>	<b>number</b>
<b>System.String</b>	<b>std::string</b>	<b>string</b>
<b>System.Byte[]</b>	<b>proton::binary</b>	<b>string</b>
<b>System.SByte</b>	<b>int8_t</b>	<b>number</b>
<b>System.Int16</b>	<b>int16_t</b>	<b>number</b>
<b>System.Int32</b>	<b>int32_t</b>	<b>number</b>
<b>System.Int64</b>	<b>int64_t</b>	<b>number</b>
<b>System.Byte</b>	<b>uint8_t</b>	<b>number</b>
<b>System.UInt16</b>	<b>uint16_t</b>	<b>number</b>
<b>System.UInt32</b>	<b>uint32_t</b>	<b>number</b>
<b>System.UInt64</b>	<b>uint64_t</b>	<b>number</b>

AMQ .NET type before encoding	AMQ C++ type	AMQ JavaScript type
<b>System.Single</b>	<b>float</b>	<b>number</b>
<b>System.Double</b>	<b>double</b>	<b>number</b>
<b>Amqp.List</b>	<b>std::vector</b>	<b>Array</b>
<b>Amqp.Map</b>	<b>std::map</b>	<b>object</b>
<b>System.Guid</b>	<b>proton::uuid</b>	<b>number</b>
<b>Amqp.Symbol</b>	<b>proton::symbol</b>	<b>string</b>
<b>System.DateTime</b>	<b>proton::timestamp</b>	<b>number</b>

Table 6.4. AMQ .NET and other AMQ client types (2 of 2)

AMQ .NET type before encoding	AMQ Python type	AMQ Ruby type
<b>null</b>	<b>None</b>	<b>nil</b>
<b>System.Boolean</b>	<b>bool</b>	<b>true, false</b>
<b>System.Char</b>	<b>unicode</b>	<b>String</b>
<b>System.String</b>	<b>unicode</b>	<b>String</b>
<b>System.Byte[]</b>	<b>bytes</b>	<b>String</b>
<b>System.SByte</b>	<b>int</b>	<b>Integer</b>
<b>System.Int16</b>	<b>int</b>	<b>Integer</b>
<b>System.Int32</b>	<b>long</b>	<b>Integer</b>
<b>System.Int64</b>	<b>long</b>	<b>Integer</b>
<b>System.Byte</b>	<b>long</b>	<b>Integer</b>
<b>System.UInt16</b>	<b>long</b>	<b>Integer</b>
<b>System.UInt32</b>	<b>long</b>	<b>Integer</b>
<b>System.UInt64</b>	<b>long</b>	<b>Integer</b>

AMQ .NET type before encoding	AMQ Python type	AMQ Ruby type
<b>System.Single</b>	<b>float</b>	<b>Float</b>
<b>System.Double</b>	<b>float</b>	<b>Float</b>
<b>Amqp.List</b>	<b>list</b>	<b>Array</b>
<b>Amqp.Map</b>	<b>dict</b>	<b>Hash</b>
<b>System.Guid</b>	-	-
<b>Amqp.Symbol</b>	<b>str</b>	<b>Symbol</b>
<b>System.DateTime</b>	<b>long</b>	<b>Time</b>

## 6.2. 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 .NET 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 6.5. AMQ .NET and JMS message types

AMQ .NET body type	JMS message type
<b>System.String</b>	<b>TextMessage</b>
<b>null</b>	<b>TextMessage</b>
<b>System.Byte[]</b>	<b>BytesMessage</b>
Any other type	<b>ObjectMessage</b>

## 6.3. CONNECTING TO AMQ BROKER

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

- Port 5672 in the network firewall is open.
- The AMQ Broker AMQP acceptor is enabled. See [Default acceptor settings](#).

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

## 6.4. CONNECTING TO AMQ INTERCONNECT

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

- Port 5672 in the network firewall is open.
- The router is configured to permit access from your client, and the client is configured to send the required credentials. See [Securing network connections](#).

## APPENDIX A. MANAGING CERTIFICATES

### A.1. INSTALLING CERTIFICATE AUTHORITY CERTIFICATES

SSL/TLS authentication relies on digital certificates issued by trusted Certificate Authorities (CAs). When an SSL/TLS connection is established by a client, the AMQP peer sends a server certificate to the client. This server certificate must be signed by one of the CAs in the client's *Trusted Root Certification Authorities* certificate store.

If the user is creating self-signed certificates for use by Red Hat AMQ Broker, then the user must create a CA to sign the certificates. Then the user can enable the client SSL/TLS handshake by installing the self-signed CA file **ca.crt**.

1. From an administrator command prompt, run the MMC Certificate Manager plugin, **certmgr.msc**.
2. Expand the **Trusted Root Certification Authorities** folder on the left to expose **Certificates**.
3. Right-click **Certificates** and select **All Tasks** and then **Import**.
4. Click **Next**.
5. Browse to select file **ca.crt**.
6. Click **Next**.
7. Select **Place all certificates in the following store**
8. Select certificate store **Trusted Root Certification Authorities**.
9. Click **Next**.
10. Click **Finish**.

For more information about installing certificates, see [Managing Microsoft Certificate Services and SSL](#).

### A.2. INSTALLING CLIENT CERTIFICATES

In order to use SSL/TLS and client certificates, the certificates with the client's private keys must be imported into the proper certificate store on the client system.

1. From an administrator command prompt, run the MMC Certificate Manager plugin, **certmgr.msc**.
2. Expand the **Personal** folder on the left to expose **Certificates**.
3. Right-click **Certificates** and select **All Tasks** and then **Import**.
4. Click **Next**.
5. Click **Browse**.
6. In the file type pulldown, select **Personal Information Exchange (\*.pfx;\*.p12)**.
7. Select file **client.p12** and click **Open**.

8. Click **Next**.
9. Enter the password for the private key password field. Accept the default import options.
10. Click **Next**.
11. Select **Place all certificates in the following store**
12. Select certificate store **Personal**.
13. Click **Next**.
14. Click **Finish**.

### A.3. HELLO WORLD USING CLIENT CERTIFICATES

Before a client will return a certificate to the broker, the AMQ .NET library must be told which certificates to use. The client certificate file **client.crt** is added to the list of certificates to be used during **SChannel** connection startup.

```
factory.SSL.ClientCertificates.Add(  
    X509Certificate.CreateFromCertFile(certfile)  
);
```

In this example, **certfile** is the full path to the **client.p12** certificate installed in the *Personal* certificate store. A complete example is found in **HelloWorld-client-certs.cs**. This source file and the supporting project files are available in the SDK.

## APPENDIX B. EXAMPLE PROGRAMS

### B.1. PREREQUISITES

- Red Hat AMQ Broker with queue named **amq.topic** and with a queue named **service\_queue** both with read/write permissions. For this illustration the broker was at IP address **10.10.1.1**.
- Red Hat AMQ Interconnect with source and target name **amq.topic** with suitable permissions. For this illustration the router was at IP address **10.10.2.2**.

All the examples run from `<install-dir>\bin\Debug`.

### B.2. HELLOWORLD SIMPLE

HelloWorld-simple is a simple example that creates a Sender and a Receiver for the same address, sends a message to the address, reads a message from the address, and prints the result.

#### HelloWorld-simple command line options

```
Command line:  
HelloWorld-simple [brokerUrl [brokerEndpointAddress]]  
Default:  
HelloWorld-simple amqp://localhost:5672 amq.topic
```

#### HelloWorld-simple sample invocation

```
$ HelloWorld-simple  
Hello world!
```

By default, this program connects to a broker running on localhost:5672. Specify a host and port, and the AMQP endpoint address explicitly on the command line:

```
$ HelloWorld-simple amqp://someotherhost.com:5672 endpointname
```

By default, this program addresses its messages to **amq.topic**. In some Amqp brokers amq.topic is a predefined endpoint address and is immediately available with no broker configuration. If this address does not exist in the broker then use a broker management tool to create it.

### B.3. HELLOWORLD ROBUST

HelloWorld-robust shares all the features of the simple example with additional options and processing:

- Accessing message properties beyond the simple payload:
  - Header
  - DeliveryAnnotations
  - MessageAnnotations
  - Properties
  - ApplicationProperties



- BodySection
- Footer
- Connection shutdown sequence

### HelloWorld-robust command line options

Command line:

```
HelloWorld-robust [brokerUrl [brokerEndpointAddress [payloadText [enableTrace]]]]
```

Default:

```
HelloWorld-robust amqp://localhost:5672 amq.topic "Hello World"
```



#### NOTE

The simple presence of the *enableTrace* argument enables tracing. The argument may hold any value.

### HelloWorld-robust sample invocation

```
$ HelloWorld-robust
```

```
Broker: amqp://localhost:5672, Address: amq.topic, Payload: Hello World!  
body:Hello World!
```

HelloWorld-robust allows the user to specify a payload string and to enable trace protocol logging.

```
$ HelloWorld-robust amqp://localhost:5672 amq.topic "My Hello" loggingOn
```

## B.4. INTEROP.DRAIN.CS, INTEROP.SPOUT.CS (PERFORMANCE EXERCISER)

AMQ .NET examples *Interop.Drain* and *Interop.Spout* illustrate interaction with Red Hat AMQ Interconnect. In this case there is no message broker. Instead the Red Hat AMQ Interconnect registers the addresses requested by the client programs and routes messages between them.

### Interop.Drain command line options

```
$ Interop.Drain.exe --help
```

```
Usage: interop.drain [OPTIONS] --address STRING
```

Create a connection, attach a receiver to an address, and receive messages.

Options:

```
--broker [amqp://guest:guest@127.0.0.1:5672] - AMQP 1.0 peer connection address
```

```
--address STRING [] - AMQP 1.0 terminus name
```

```
--timeout SECONDS [1] - time to wait for each message to be received
```

```
--forever [false] - use infinite receive timeout
```

```
--count INT [1] - receive this many messages and exit; 0 disables count based exit
```

```
--initial-credit INT [10] - receiver initial credit
```

```
--reset-credit INT [5] - reset credit to initial-credit every reset-credit messages
```

```
--quiet [false] - do not print each message's content
```

```
--help - print this message and exit
```

Exit codes:

- 0 - successfully received all messages
- 1 - timeout waiting for a message
- 2 - other error

### Interop.Spout command line options

```
$ interop.spout --help
Usage: Interop.Spout [OPTIONS] --address STRING
Create a connection, attach a sender to an address, and send messages.

Options:
--broker [amqp://guest:guest@127.0.0.1:5672] - AMQP 1.0 peer connection address
--address STRING [] - AMQP 1.0 terminus name
--timeout SECONDS [0] - send for N seconds; 0 disables timeout
--durable [false] - send messages marked as durable
--count INT [1] - send this many messages and exit; 0 disables count based exit
--id STRING [guid] - message id
--replyto STRING [] - message ReplyTo address
--content STRING [] - message content
--print [false] - print each message's content
--help - print this message and exit

Exit codes:
0 - successfully received all messages
2 - other error
```

### Interop.Spout and Interop.Drain sample invocation

In one window run Interop.drain. Drain waits forever for one message to arrive.

```
$ Interop.Drain.exe --broker amqp://10.10.2.2:5672 --forever --count 1 --address amq.topic
```

In another window run Interop.spout. Spout sends a message to the broker address and exits.

```
$ interop.spout --broker amqp://10.10.2.2:5672 --address amq.topic
$
```

Now in the first window drain will have received the message from spout and then exited.

```
$ Interop.Drain.exe --broker amqp://10.10.2.2:5672 --forever --count 1 --address amq.topic
Message(Properties=properties(message-id:9803e781-14d3-4fa7-8e39-c65e18f3e8ea:0),
ApplicationProperties=, Body=
$
```

## B.5. INTEROP.CLIENT, INTEROP.SERVER (REQUEST-RESPONSE)

This example shows a simple broker-based server that will accept strings from a client, convert them to upper case, and send them back to the client. It has two components:

- client - sends lines of poetry to the server and prints responses.
- server - a simple service that will convert incoming strings to upper case and return them to the requester.

In this example the server and client share a service endpoint in the broker named **service\_queue**. The

server listens for messages at the service endpoint. Clients create temporary dynamic ReplyTo queues, embed the temporary name in the requests, and send the requests to the server. After receiving and processing each request the server sends the reply to the client's temporary ReplyTo address.

### Interop.Client command line options

```
Command line:
  Interop.Client [peerURI [loopcount]]
Default:
  Interop.Client amqp://guest:guest@localhost:5672 1
```

### Interop.Server command line options

```
Command line:
  Interop.Server [peerURI]
Default:
  Interop.Server amqp://guest:guest@localhost:5672
```

### Interop.Client, Interop.Server sample invocation

The programs may be launched with these command lines:

```
$ Interop.Server.exe amqp://guest:guest@localhost:5672
$ Interop.Client.exe amqp://guest:guest@localhost:5672
```

PeerToPeer.Server creates a listener on the address given in the command line. This address initializes a *ContainerHost* class object that listens for incoming connections. Received messages are forwarded asynchronously to a *RequestProcessor* class object.

PeerToPeer.Client opens a connection to the server and starts sending messages to the server.

### PeerToPeer.Client command line options

```
Command line:
  PeerToPeer.Client [peerURI]
Default:
  PeerToPeer.Client amqp://guest:guest@localhost:5672
```

### PeerToPeer.Server command line options

```
Command line:
  PeerToPeer.Server [peerURI]
Default:
  PeerToPeer.Server amqp://guest:guest@localhost:5672
```

### PeerToPeer.Client, PeerToPeer.Server sample invocation

In one window run the PeerToPeer.Server

```
$ PeerToPeer.Server.exe
Container host is listening on 127.0.0.1:5672
Request processor is registered on request_processor
Press enter key to exist...
Received a request hello 0
...
```

In another window run PeerToPeer.Client. PeerToPeer.Client sends messages the the server and prints responses as they are received.

```
$ PeerToPeer.Client.exe
Running request client...
Sent request properties(message-id:command-request,reply-to:client-57db8f65-6e3d-474c-a05e-
8ca63b69d7c0) body hello 0
Received response: body reply0
Received response: body reply1
^C
```

## APPENDIX C. 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](https://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](https://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](https://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](https://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 2020-02-26 17:14:27 UTC*