Red Hat AMQ Clients 2.11

Using the AMQ .NET Client

For Use with AMQ Clients 2.11

Last Updated: 2023-08-30

Red Hat AMQ Clients 2.11 Using the AMQ .NET Client

For Use with AMQ Clients 2.11

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

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. OVERVIEW

AMQ .NET is a lightweight AMQP 1.0 library for the .NET platform. 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 2.11 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 1.0, 1.1, 1.2, and 1.3 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

Refer to Red Hat AMQ Supported Configurations on the Red Hat Customer Portal for current information regarding AMQ .NET 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
Connection	A channel for communication between two peers on a network
Session	A context for sending and receiving messages

Entity	Description
Sender link	A channel for sending messages to a target
Receiver link	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

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.

1.5. DOCUMENT CONVENTIONS

The sudo command

In this document, **sudo** is used for any command that requires root privileges. Exercise caution when using **sudo** because any changes can affect the entire system. For more information about **sudo**, see Using the sudo command.

File paths

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

Variable text

This document contains code blocks with variables that you must replace with values specific to your environment. Variable text is enclosed in arrow braces and styled as italic monospace. For example, in the following command, replace ***<project-dir>*** with the value for your environment:

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

CHAPTER 2. INSTALLATION

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

2.1. PREREQUISITES

- You must have a subscription to access AMQ release files and repositories.
- To use AMQ .NET on Red Hat Enterprise Linux, you must install the the .NET Core 3.1 developer tools. For information, see the .NET Core 3.1 getting started guide.
- To build programs using AMQ .NET on Microsoft Windows, you must install Visual Studio.

2.2. INSTALLING ON RED HAT ENTERPRISE LINUX

Procedure

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 **INTEGRATION AND AUTOMATION** category.
3. Click **Red Hat AMQ Clients** The **Software Downloads** page opens.
4. Download the **AMQ Clients 2.11.0 .NET Core** .zip file.
5. Use the **unzip** command to extract the file contents into a directory of your choosing.

   ```
   $ unzip amq-clients-2.11.0-dotnet-core.zip
   ```

   When you extract the contents of the .zip file, a directory named **amq-clients-2.11.0-dotnet-core** is created. This is the top-level directory of the installation and is referred to as ***<install-dir>*** throughout this document.
6. Use a text editor to create the file **$HOME/.nuget/NuGet/NuGet.Config** and add the following content:

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <configuration>
     <packageSources>
       <add key="nuget.org" value="https://api.nuget.org/v3/index.json" protocolVersion="3"/>
       <add key="amq-clients" value="<install-dir>/nupkg"/>
     </packageSources>
   </configuration>
   ```

   If you already have a **NuGet.Config** file, add the **amq-clients** line to it.

   Alternatively, you can move the .nupkg file inside the ***<install-dir>*/nupkg** directory to an existing package source location.

2.3. INSTALLING ON MICROSOFT WINDOWS

Procedure

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 **INTEGRATION AND AUTOMATION** category.
3. Click **Red Hat AMQ Clients** The **Software Downloads** page opens.
4. Download the **AMQ Clients 2.11.0 .NET** .zip file.
5. Extract the file contents into a directory of your choosing by right-clicking on the zip file and selecting **Extract All**.

When you extract the contents of the .zip file, a directory named **amq-clients-2.11.0-dotnet** is created. This is the top-level directory of the installation and is referred to as ***<install-dir>*** throughout this document.

CHAPTER 3. GETTING STARTED

This chapter guides you through the steps to set up your environment and run a simple messaging program.

3.1. PREREQUISITES

- You must complete the installation procedure for your environment.
- You must have an AMQP 1.0 message broker listening for connections on interface **localhost** and port **5672**. It must have anonymous access enabled. For more information, see Starting the broker.
- You must have a queue named **amq.topic**. For more information, see Creating a queue.

3.2. RUNNING HELLOWORLD ON RED HAT ENTERPRISE LINUX

The Hello World example creates a connection to the broker, sends a message containing a greeting to the **amq.topic** queue, and receives it back. On success, it prints the received message to the console.

Change to the ***<install-dir>*/examples/netcoreapp3/HelloWorld-simple** and use **dotnet run** to build and execute the program.

```
$ cd <install-dir>/examples/netcoreapp3/HelloWorld-simple
$ dotnet run
Hello World!
```

3.3. RUNNING HELLO WORLD ON MICROSOFT WINDOWS

The Hello World example creates a connection to the broker, sends a message containing a greeting to the **amq.topic** queue, and receives it back. On success, it prints the received message to the console.

Procedure

1. Navigate to ***<install-dir>*** and open the **amqp.sln** solution file in Visual Studio.
2. Select **Build Solution** from the **Build** menu to compile the solution.
3. Open a command prompt window and execute the following commands to send and receive a message:

```
> cd <install-dir>\bin\Debug
> 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 and the AMQP.Net Lite examples.

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

To run the example program, compile it and execute it from the command line. For more information, see Chapter 3, *Getting started*.

```
<install-dir>\bin\Debug>simple_send "amqp://guest:guest@localhost" service_queue
```

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";      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);
            ReceiverLink receiver = new ReceiverLink(session, "recv-1", source);7
```

```
            for (int i = 0; i < count; i++)
            {
                Message msg = receiver.Receive();                    8
                receiver.Accept(msg);                                9
                Console.WriteLine("Received: " + msg.Body.ToString());
            }

            receiver.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] **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 *recv-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

To run the example program, compile it and execute it from the command line. For more information, see Chapter 3, *Getting started*.

```
<install-dir>\bin\Debug>simple_recv "amqp://guest:guest@localhost" service_queue
```

CHAPTER 5. NETWORK CONNECTIONS

5.1. CONNECTION URIS

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. RECONNECT AND FAILOVER

AMQ .NET does not offer reconnect and failover, but it can be implemented in your application by intercepting connection errors and reconnecting. For example code, see the ReconnectSender.cs example.

CHAPTER 6. SECURITY

6.1. 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);
```

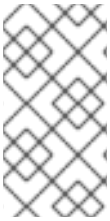
6.2. CONFIGURING SASL AUTHENTICATION

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.

6.3. 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.

CHAPTER 7. 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.

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

7.1. 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

7.2. 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.

7.3. 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.

CHAPTER 8. MESSAGE DELIVERY

8.1. 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.

8.2. 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.

CHAPTER 9. LOGGING

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

9.1. SETTING THE LOG OUTPUT LEVEL

The library emits log traces at different levels:

- Error
- Warning
- Information
- Verbose

The lowest log level, *Error*, traces only error events and produces 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
```

9.2. 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 logically 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 10. 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.

10.1. 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 10.1. 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 10.2. AMQ .NET types before encoding and after decoding

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

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

Table 10.3. AMQ .NET and other AMQ client types (1 of 2)

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

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

Table 10.4. AMQ .NET and other AMQ client types (2 of 2)

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

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

10.2. 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 10.5. AMQ .NET and JMS message types

AMQ .NET body type	JMS message type
System.String	**TextMessage**
null	**TextMessage**
System.Byte[]	**BytesMessage**
Any other type	**ObjectMessage**

10.3. 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.

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 certficates, 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.

C.1. ACCESSING YOUR ACCOUNT

Procedure

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

C.2. ACTIVATING A SUBSCRIPTION

Procedure

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

C.3. DOWNLOADING RELEASE FILES

To access .zip, .tar.gz, and other release files, use the customer portal to find the relevant files for download. If you are using RPM packages or the Red Hat Maven repository, this step is not required.

Procedure

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 **INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

C.4. REGISTERING YOUR SYSTEM FOR PACKAGES

To install RPM packages for this product on Red Hat Enterprise Linux, your system must be registered. If you are using downloaded release files, this step is not required.

Procedure

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.

For more information about registering your system, see one of the following resources:

- Red Hat Enterprise Linux 7 - Registering the system and managing subscriptions
- Red Hat Enterprise Linux 8 - Registering the system and managing subscriptions

APPENDIX D. USING AMQ BROKER WITH THE EXAMPLES

The AMQ .NET examples require a running message broker with a queue named **amq.topic**. Use the procedures below to install and start the broker and define the queue.

D.1. INSTALLING THE BROKER

Follow the instructions in *Getting Started with AMQ Broker* to install the broker and create a broker instance. Enable anonymous access.

The following procedures refer to the location of the broker instance as ***<broker-instance-dir>***.

D.2. STARTING THE BROKER

Procedure

1. Use the **artemis run** command to start the broker.

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

2. Check the console output for any critical errors logged during startup. The broker logs **Server is now live** when it is ready.

```
$ 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
...
```

D.3. CREATING A QUEUE

In a new terminal, use the **artemis queue** command to create a queue named **amq.topic**.

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

You are prompted to answer a series of yes or no questions. Answer **N** for no to all of them.

Once the queue is created, the broker is ready for use with the example programs.

D.4. STOPPING THE BROKER

When you are done running the examples, use the **artemis stop** command to stop the broker.

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

Revised on 2023-08-30 13:02:29 UTC