RabbitMQ .NET Client Library User Guide

Copyright © 2007, 2008, 2009, 2010, 2011 VMware, Inc.

Table of Contents

1. UserGuide

1.1. Introduction

The RabbitMQ .NET client is an implementation of an AMQP client library for C# (and, implicitly, other .NET languages), and a binding exposing AMQP services via Microsoft's Windows Communication Foundation (WCF).

This is the User Guide for the RabbitMQ .NET client library. It provides an overview of the codebase and the implemented API.

The full details of the API are documented separately, in the NDocProc-generated javadoc-like HTML documentation.

1.2. AMQP dialects implemented

The client library implements AMQP specifications version 0-8, 0-8bis (0-8 as modified by QPid for their M2 release), 0-9 (omitting sections of the specification marked "work in progress", i.e. the `Message` content-class) and 0-9-1. The ApiGen tool processes the specification XML files in conjunction with certain C# interfaces, generating C# code directly, which is compiled into the final assembly.

2. ApiOverview

This section gives an overview of the RabbitMQ .NET client API.

Only the basics of using the library are covered: for full detail, please see the javadoc-like API documentation generated from the source code.

2.1. Major namespaces, interfaces and classes

The API is closely modelled on the AMQP protocol specification, with little additional abstraction.

The core API interfaces and classes are defined in the `RabbitMQ.Client` namespace:

```
using RabbitMQ.Client;
```

The core API interfaces and classes are

- `IModel`: represents an AMQP data channel, and provides most of the AMQP operations
- `IConnection`: represents an AMQP connection
- `ConnectionFactory`: constructs `IConnection` instances

Other useful interfaces and classes include:

- `ConnectionParameters`: configures a `ConnectionFactory`
- `QueueingBasicConsumer`: receives messages delivered from the server
- `Protocols`: support class for choosing an AMQP protocol variant

Public namespaces other than `RabbitMQ.Client` include:

- `RabbitMQ.Client.Content`: construction and analysis of messages that are binary-compatible with messages produced and consumed by QPid's JMS compatibility layer.
- `RabbitMQ.Client.Events`: various events and event handlers that are part of the AMQP client library.
- `RabbitMQ.Client.Exceptions`: exceptions visible to the user of the AMQP client library.

All other namespaces are reserved for private implementation detail of the library, although members of private namespaces are usually made available to applications using the library in order to permit developers to implement workarounds for faults or design mistakes they discover in the library implementation. Applications cannot rely on any classes, interfaces, member variables etc. that appear within private namespaces remaining stable across releases of the library.

2.2. Multi-protocol support

Class `RabbitMQ.Client.Protocols` contains convenient predefined `IProtocol` instances that permit selection of a supported protocol variant. At the time of writing, the library supports:

- `Protocols.DefaultProtocol`: An alias to one of the other protocols. At the time of writing, *the default protocol is AMQP_0_9_1*.
- `Protocols.AMQP_0_8`: Standard AMQP 0-8 protocol, including `access.request`, `basic.recover-async` and `queue.unbind`.
- `Protocols.AMQP_0_8_QPID`: AMQP 0-8 as extended by QPid's 0-8 M1 release, without `access.request` and with extra parameters available on certain methods.
- `Protocols.AMQP_0_9`: Standard AMQP 0-9 protocol, including `access.request` and `basic.recover-async`, but excluding the `Message` content-class and other areas of the specification marked "work in progress".
- `Protocols.AMQP_0_9_1`: Standard AMQP 0-9-1 protocol.

The `Protocols` class also contains some convenience functions for retrieving an IProtocol from a string, from the environment or from a .NET XML configuration file. The protocol names permitted are the same as the static instance variables on class `Protocols`, that is, `AMQP_0_8`, `AMQP_0_8_QPID`, `AMQP_0_9`, `AMQP_0_9_1` and `DefaultProtocol`.

The following code binds `p` to the `Protocols.AMQP_0_9` instance by looking it up dynamically:

```
IProtocol p = Protocols.SafeLookup("AMQP_0_9");
```

Given the following `App.config` snippet,

```
<appSettings>
  <add key="my-protocol" value="AMQP_0_9"/>
</appSettings>
```

the following code will also bind `p` to the 0-9 `IProtocol` implementation:

```
IProtocol p = Protocols.FromConfiguration("my-protocol");
```

An alternative is `Protocols.FromEnvironmentVariable()`, which reads the name of the protocol variant to return from the shell environment variable named by `Protocols.EnvironmentVariable` (at the time of writing, `AMQP_PROTOCOL`).

Finally, `Protocols.FromEnvironment()` tries `FromConfiguration()` first, and if no setting is found, falls back to `FromEnvironmentVariable()`.

If no argument is passed to `FromEnvironment()` or `FromConfiguration()`, the value of `Protocols.DefaultAppSettingsKey` is used. (At the time of writing, the default appSettings key is `AMQP_PROTOCOL`, the same as the name of the shell environment variable scanned.)

2.3. Connecting to a broker

The following code connects to an AMQP broker:

```
ConnectionFactory factory = new ConnectionFactory();
factory.UserName = user;
factory.Password = pass;
factory.VirtualHost = vhost;
factory.Protocol = Protocols.FromEnvironment();
factory.HostName = hostName;
factory.Port = AmqpTcpEndpoint.UseDefaultPort;
IConnection conn = factory.CreateConnection();
```

All factory properties have default values. The default value for a property will be used if the property remains unassigned prior to creating a connection:

- username: `"guest"`
- password: `"guest"`
- virtual-host: `"/"`
- protocol: `AMQP 0-9-1`
- hostname: `localhost`
- port: `5672`

The IConnection interface can then be used to open a channel:

```
IModel channel = conn.CreateModel();
```

The channel can now be used to send and receive messages, as described in subsequent sections.

2.4. Disconnecting from a broker

To disconnect, simply close the channel and the connection:

```
channel.Close(200, "Goodbye");
conn.Close();
```

Note that closing the channel is considered good practice, but isn't strictly necessary - it will be done automatically anyway when the underlying connection is closed.

In some situations, you may want the connection to close automatically once the last open channel on the connection closes. To achieve this, set the `IConnection.AutoClose` property to true, but only *after* creating the first channel:

```
IConnection conn = factory.CreateConnection(...);
IModel channel = conn.CreateModel();
conn.AutoClose = true;
```

When `AutoClose` is true, the last channel to close will also cause the connection to close[1]. If it is set to true before any channel is created, the connection will close then and there.

2.5. Using exchanges and queues

Client applications work with exchanges and queues, the high-level building blocks of AMQP. These must be "declared" before they can be used. Declaring either type of object simply ensures that one of that name exists, creating it if necessary.

Continuing the previous example, the following code declares an exchange and a queue, then binds them together.

```
channel.ExchangeDeclare(exchangeName, ExchangeType.Direct);
channel.QueueDeclare(queueName, false, false, false, null);
channel.QueueBind(queueName, exchangeName, routingKey, null);
```

This will actively declare the following objects:

1. a non-durable, non-autodelete exchange of "direct" type

2. a non-durable, non-exclusive, non-autodelete queue

The exchange can be customised by using additional parameters. The above code then binds the queue to the exchange with the given routing key.

Note that many Channel API methods are overloaded. The convenient short form of ExchangeDeclare uses sensible defaults. There are also longer forms with more parameters, to let you override these defaults as necessary, giving full control where needed.

This "short version, long version" pattern is used throughout the API.

2.6. Publishing messages

To publish a message to an exchange, use `IModel.BasicPublish` as follows:

```
byte[] messageBodyBytes = System.Text.Encoding.UTF8.GetBytes("Hello, world!");
channel.BasicPublish(exchangeName, routingKey, null, messageBodyBytes);
```

For fine control, you can use overloaded variants to specify the mandatory and immediate flags, or send messages with basic-class header properties:

```
byte[] messageBodyBytes = System.Text.Encoding.UTF8.GetBytes("Hello, world!");
IBasicProperties props = channel.CreateBasicProperties();
props.ContentType = "text/plain";
props.DeliveryMode = 2;
channel.BasicPublish(exchangeName,
                     routingKey, props,
                     messageBodyBytes);
```

This sends a message with delivery mode 2 (persistent) and content-type "text/plain". See the definition of the `IBasicProperties` interface for more information about the available header properties.

2.7. Retrieving individual messages

To retrieve individual messages, use `IModel.BasicGet`. The returned value is an instance of BasicGetResult, from which the header information (properties) and message body can be extracted:

```
bool noAck = false;
BasicGetResult result = channel.BasicGet(queueName, noAck);
if (result == null) {
    // No message available at this time.
} else {
    IBasicProperties props = result.BasicProperties;
    byte[] body = result.Body;
    ...
```

Since `noAck = false` above, you must also call `IModel.BasicAck` to acknowledge that you have successfully received and processed the message:

```
    ...
    // acknowledge receipt of the message
    channel.BasicAck(result.DeliveryTag, false);
}
```

2.8. Retrieving messages by subscription

Another way to receive messages is to set up a subscription using the `IBasicConsumer` interface. The messages will then be delivered automatically as they arrive, rather than having to be requested proactively.

The easiest and safest way to implement a consumer is to use the convenience class `QueueingBasicConsumer`, and retrieve deliveries from the `SharedQueue` instance contained therein:

```
QueueingBasicConsumer consumer = new QueueingBasicConsumer(channel);
String consumerTag = channel.BasicConsume(queueName, false, consumer);
while (true) {
    try {
        RabbitMQ.Client.Events.BasicDeliverEventArgs e =
          (RabbitMQ.Client.Events.BasicDeliverEventArgs)
          consumer.Queue.Dequeue();
        IBasicProperties props = e.BasicProperties;
        byte[] body = e.Body;
        // ... process the message
        channel.BasicAck(e.DeliveryTag, false);
    } catch (OperationInterruptedException ex) {
        // The consumer was removed, either through
        // channel or connection closure, or through the
        // action of IModel.BasicCancel().
        break;
```

```
    }
}
```

Another alternative is to subclass `DefaultBasicConsumer`, overriding methods as necessary, or implement `IBasicConsumer` directly. You will generally want to implement the core method `HandleBasicDeliver`.

More sophisticated consumers will need to implement further methods. In particular, `HandleModelShutdown` traps channel/connection closure. Consumers can also implement `HandleBasicCancelOk` to be notified of cancellations.

The `ConsumerTag` property of `DefaultBasicConsumer` can be used to retrieve the server-generated consumer tag, in cases where none was supplied to the original `IModel.BasicConsume` call.

You can cancel an active consumer with `IModel.BasicCancel`:

```
channel.BasicCancel(consumerTag);
```

When calling the API methods, you always refer to consumers by their consumer tags, which can be either client- or server-generated as explained in the AMQP specification document.

2.9. Threading, deadlocks, and associated restrictions on consumers

Each `IConnection` instance is, in the current implementation, backed by a single background thread that reads from the socket and dispatches the resulting events to the application. Additionally there are two threads responsible for `Heartbeat` support, if connection negotiation turned it on.

Usually, therefore, there will be four threads active in an application using this library:

- the application thread: contains the application logic, and makes calls on `IModel` methods to perform AMQP operations.
- the connection thread: hidden away and completely managed by the `IConnection` instance.
- the heartbeat read thread: continuously checking if `IConnection` instance has not missed too many `Heartbeat` frames from the broker
- the heartbeat write thread: continuosuly checking if `IConnection` instance should send `Heartbeat` frame to the broker

The one place where the nature of the threading model is visible to the application is in any callback the application registers with the library. Such callbacks include:

- any `IBasicConsumer` method

- the `BasicReturn` event on `IModel`

- any of the various shutdown events on `IConnection`, `IModel` etc.

Application callback handlers *must not* invoke blocking AMQP operations (such as `IModel.QueueDeclare`, `IModel.BasicCancel` or `IModel.BasicPublish`). If they do, the channel will deadlock[2].

Only asynchronous AMQP operations without content are safe for use within callbacks, such as `IModel.BasicAck`. It is not safe to use `IModel.BasicPublish` in callbacks, because channel flow could block publication.

For this reason, `QueueingBasicConsumer` is the safest way of subscribing to a queue, because its implementation uses `RabbitMQ.Util.SharedQueue` to pass deliveries over to the application thread, where all processing of received deliveries is done, and where any AMQP `IModel` operation is safe.

2.10. IModel should not be shared between threads

In general, `IModel` instances should not be used by more than one thread simultaneously: application code should maintain a clear notion of thread ownership for `IModel` instances.

If more than one thread needs to access a particular `IModel` instances, the application should enforce mutual exclusion itself. One way of achieving this is for all users of an `IModel` to `lock` the instance itself:

```
IModel ch = RetrieveSomeSharedIModelInstance();
lock (ch) {
  ch.BasicPublish(...);
}
```

Symptoms of incorrect serialisation of `IModel` operations include, but are not limited to,

- invalid frame sequences being sent on the wire (which occurs, for example, if more than one `BasicPublish` operation is run simultaneously), and/or

- `NotSupportedException`s being thrown from a method in class `RpcContinuationQueue` complaining about `"Pipelining of requests forbidden"` (which occurs in situations where more than one AMQP RPC, such as `ExchangeDeclare`, is run simultaneously).

2.11. Handling unroutable or undelivered messages

If a message is published with the "mandatory" or "immediate" flags set, but cannot be delivered, the broker will return it to the sending client (via a `basic.return` AMQP command).

To be notified of such returns, clients can subscribe to the `IModel.BasicReturn` event. If there are no listeners attached to the event, then returned messages will be silently dropped.

```
channel.BasicReturn +=
  new RabbitMQ.Client.Events.BasicReturnEventHandler(...);
```

The `BasicReturn` event will fire, for example, if the client publishes a message with the "mandatory" flag set to an exchange of "direct" type which is not bound to a queue.

2.12. QPid-compatible JMS BytesMessage, StreamMessage, and MapMessage codecs

The `RabbitMQ.Client.Content` namespace contains classes and interfaces that implement QPid-compatible JMS BytesMessage, StreamMessage and MapMessage encoders and decoders.

For example, to construct (and send) a MapMessage:

```
IModel channel = ...;

IMapMessageBuilder b = new MapMessageBuilder(channel);
b.Headers["header1"] = "some@random.string";
b.Body["field1"] = 123.45;
b.Body["field2"] = new byte[] { 1, 2, 3 };

channel.BasicPublish(exchange, routingKey,
                     (IBasicProperties)  b.GetContentHeader(),
                     b.GetContentBody());
```

A StreamMessage is similar (and BytesMessages are so similar to StreamMessages that I shan't demonstrate them here):

```
IStreamMessageBuilder b = new StreamMessageBuilder(channel);
b.Headers["header1"] = "some@random.string";
b.WriteDouble(123.45);
b.WriteBytes(new byte[] { 1, 2, 3 });
```

Given a received message in the form of an `IBasicProperties` header and a `byte[]` body, reading the message as a MapMessage is done with `IMapMessageReader`:

```
IBasicProperties props = ...;
byte[] receivedBody = ...;
IMapMessageReader r = new MapMessageReader(props, receivedBody);
Console.Out.WriteLine("Header1: {0}", r.Headers["header1"]);
Console.Out.WriteLine("Field1: {0}", r.Body["field1"]);
Console.Out.WriteLine("Field2: {0}", r.Body["field2"]);
```

StreamMessages (and BytesMessages) are similar:

```
IStreamMessageReader r = new StreamMessageReader(props, receivedBody);
Console.Out.WriteLine("Header1: {0}", r.Headers["header1"]);
Console.Out.WriteLine("First value: {0}", r.ReadDouble());
Console.Out.WriteLine("Second value: {0}", r.ReadBytes());
```

3. MessagingPatterns

3.1. Common ways of working with AMQP

When building distributed systems with AMQP, there are a number of different messaging patterns that crop up over and over again. In this section, we cover some of the most common coding patterns and interaction styles:

- Point-to-point messaging: both remote procedure call (RPC), and asynchronous messages directed toward a particular receiver

- Event broadcasting: one-to-many interactions; transmission of messages directed implicitly to a set of interested receivers, with collection of zero or more possible responses

- Responsibility transfer: choosing which piece of the network is responsible for any given message

- Reliable message transfer: at-least-once, at-most-once and exactly-once message delivery guarantees

- Coding with high-availability AMQP server pairs

- Preserving atomicity and idempotence when interacting with external resources

Limited library support is also available for working with these patterns, in the `RabbitMQ.Client.MessagePatterns` namespace:

- `Subscription` provides a high-level interface to receiving messages from the server

- `SimpleRpcServer` builds on `Subscription` to implement an RPC or one-way service

- `SimpleRpcClient` builds on `Subscription` to interact with remote services

Future releases of the RabbitMQ .NET client library will include improved high-level support for the most common messaging patterns and their variations.

3.2. Point-to-point messaging

The point-to-point messaging pattern occurs when the publisher of a message has a particular receiving application in mind - for instance, when a RPC-style service is made available via the AMQP server, or

when an application in a workflow chain receives a work item from its predecessor and sends the transformed work item to its successor.

3.2.1. Synchronous, client-server remote procedure call (RPC)

In order to perform request/response RPC,

- some means of addressing the service must be available
- some means of receiving a reply must be available
- some means of correlating the request message to the reply message must be available

3.2.1.1. Addressing the service

Since AMQP messages are published using a pair of an exchange name and a routing key, this is sufficient for addressing a service. Using a simple exchange-name/routing-key combination permits a number of different ways to implement the service while presenting the same interface to clients. For instance, the service could be implemented as a single process consuming from a queue, and load-balancing internally, or it could be multiple processes consuming from a single queue, being handed requests round-robin style, thereby load balancing without special coding in the service logic.

Messages can be addressed to the service request queue either

- directly, using the AMQP default exchange (""); or
- indirectly, by using a service-specific exchange, which leaves the routing-key free for such purposes as method selection or additional service-specific addressing information; or
- indirectly, by using an exchange shared by multiple services, with the service name encoded in the routing key.

Using an exchange other than the default exchange permits other applications to receive copies of each request message, which can be useful for monitoring, auditing, logging and debugging.

3.2.1.2. Ensuring a service instance is listening

AMQP's Basic-class publish operation (`IModel.BasicPublish`) provides two delivery flags, "mandatory" and "immediate", which can be used to ensure various kinds of service availability at the time a request is sent by a client.

Setting the "mandatory" flag causes a request to be returned if it cannot be routed to a queue, and setting the "immediate" flag causes a request to be returned if there is no service instance ready to receive (and, implicitly, process) the message at the time of publication. Returned messages appear as `basic.return` AMQP commands, which are made visible to the application via the `IModel.BasicReturn` event on the `IModel` that was used to publish the message.

Since published messages are returned to clients via AMQP's `basic.return` method, and `basic.return` is an asynchronous negative-acknowledgement event, the *absence* of a `basic.return` for a particular message cannot be taken as a confirmation of delivery: the use of "mandatory" and "immediate" only provides a way of raising the bar, rather than eliminating failure entirely.

When using "mandatory" or "immediate", it's important to be aware that different AMQP servers implement the flags in different ways, depending on the degree of internal asynchrony and distribution that those servers exhibit. In particular, the interpretation of "mandatory" and "immediate" in situations where `Tx`-class transactions are active is underspecified in both versions 0-8 and 0-9 of the AMQP specification, and different server implementations behave in different ways.

Finally, the fact that a message was flagged "mandatory", and successfully enqueued on one or more queues, is no guarantee of its eventual receipt: most trivially, the queue could be deleted before the message is processed, but other situations, like the use of the `noAck` flag by a message consumer, can also make the guarantees provided by "mandatory" and "immediate" conditional.

3.2.1.3. Receiving replies

AMQP's Basic-class content header (`IBasicProperties`) contains a field called `ReplyTo`, which in AMQP 0-8 and 0-9 is an unstructured string that can be used to tell the service where to post a reply to a received RPC request. Across current AMQP client libraries, there are two widely-used formats for the string in the `ReplyTo` header:

- a simple queue name; or
- a URI-like string (take care - it is *not* a URI, it merely shares certain superficial characteristics with URIs) of the form

  `exchangeType://exchangeName/routingKey`

  where *exchangeType* is one of `fanout`, `direct` or `topic`, and both *exchangeName* and *routingKey* may be empty strings.

The service instance will post its reply to the named destination, and the requesting client should arrange to receive messages so addressed, using either `BasicGet` or `BasicConsume` on an appropriately-bound queue.

3.2.1.4. Correlating a received reply to a transmitted request

AMQP's Basic-class content header (`IBasicProperties`) contains a field called `CorrelationId`, which in AMQP 0-8 and 0-9 is an unstructured string that can be used to match a request to a reply. A reply message should have the same `CorrelationId` as the one that was attached to the request message.

3.2.2. Asynchronous, one-way messaging

In some situations, a simple request-reply interaction pattern is inappropriate for your application. In these cases, the interaction pattern of interest can be constructed from asynchronous, one-way, point-to-point messages.

If an application is to respond to both synchronous, RPC-style requests, and asynchronous one-way requests, it should use the value of `ReplyTo` to decide which interaction style is being requested of it: if `ReplyTo` is present and non-empty, the request can be assumed to be an RPC-style call; otherwise, it should be assumed to be a one-way message.

The `CorrelationId` field can be used to group together a number of related messages, just as for the RPC-style case, but more generally tying together an arbitrary number of messages.

3.2.3. Acknowledgment modes for point-to-point

AMQP can operate in one of two modes, when receiving messages from the server: auto-acknowledgement mode (when the `noAck` flag has been set on `BasicGet`, `BasicConsume`, or the `Subscription` constructor), or manual-acknowledgement mode. Choosing the right acknowledgement mode is important for your application:

- auto-acknowledgement mode means that the server will internally mark a message as successfully delivered as it transmits it across the network. Messages delivered in auto-acknowledgement mode will not generally be redelivered to any other receiver.
- manual-acknowledgement mode means that the server will wait for positive confirmation of receipt before marking a message as successfully delivered. Messages delivered in manual-acknowledgement mode may be redelivered arbitrarily often[3] until a receiver takes responsibility for a message by sending an acknowledgement for it.

In general,

- if a service is in manual-acknowledgement mode, it should not acknowledge the request message until it has replied to it; see the section below on interaction with external resources.
- a client may use auto-acknowledgement mode, depending on the consequences of a retransmission of the request message.

3.2.4. Library support for point-to-point messaging

The RabbitMQ .NET client library includes basic support for common tasks involving point-to-point messaging.

3.2.4.1. SimpleRpcServer

The class `RabbitMQ.Client.MessagePatterns.SimpleRpcServer` implements synchronous RPC-style request handling as well as asynchronous message handling. Users should subclass `SimpleRpcServer`, overriding one or more of the methods with names beginning with "`Handle`".

`SimpleRpcServer` instances have a request-dispatching loop, `MainLoop`, which interprets a request as an RPC-style request needing a reply if the `ReplyTo` field of the request's `IBasicProperties` is non-null and non-empty. Requests with absent or empty `ReplyTo` fields are treated as one-way.

When an RPC-style request has been processed, the reply is sent to the `ReplyTo` address. The reply address is first matched against a regular-expression describing the URI-like syntax given above; if it matches, the components of the URI-like syntax are used as the reply address, and if it does not, the whole string is used as a simple queue name, and the reply is sent to the default exchange ("") with a routing-key equal to the `ReplyTo` string.

After a reply message, including possible `IBasicProperties` headers, is produced by the service logic, the reply's `CorrelationId` is set to be equal to the `CorrelationId` of the request, if any, before transmission.

For very basic or unusual asynchronous point-to-point messaging tasks, a `Subscription` may be used instead of `SimpleRpcServer` - please see the section below on `Subscription`s.

The basic pattern for implementing a service using `SimpleRpcServer` is as follows (assuming a class `MySimpleRpcServerSubclass` extending `SimpleRpcServer`):

```
using (IConnection conn = new ConnectionFactory()
                              .CreateConnection(serverAddress)) {
    using (IModel ch = conn.CreateModel()) {

        Subscription subscription =
            new Subscription(ch, /* ... */);
        // in the line above, the "..." indicates the parameters
        // used to specify the address to use to route messages
        // to the subscription. This subscription will be used
        // to read service request messages, so the "..." effectively
        // specifies the service address.

        new MySimpleRpcServerSubclass(subscription).MainLoop();
    }
}
```

The service logic would be implemented in an override of one of the `HandleCall` or `HandleCast` overloads on `MySimpleRpcServerSubclass`. For full details, please see the code documentation for `SimpleRpcServer`.

3.2.4.2. SimpleRpcClient

The class `RabbitMQ.Client.MessagePatterns.SimpleRpcClient` implements code for interacting with `SimpleRpcServer`s or similar.

RPC-style interactions are performed with the `Call` methods. A (private) `Subscription` is set up to receive replies from the service, and the `ReplyTo` field is set to point to the subscription. The `CorrelationId` field of the request is initialised to a fresh GUID. Asynchronous/one-way interactions are simply passed on to `IModel.BasicPublish` without modification: it is up to the caller to set `CorrelationId` in the asynchronous case.

The class currently has no support for setting the "mandatory" or "immediate" flags on a published request message, nor for handling any `BasicReturn` events that might arise from setting either flag.

The code that retrieves replies from the internal `Subscription` currently cannot deal with multiple simultaneously outstanding RPC requests, because it requires that replies arrive in the same order as the requests were sent out. Do not attempt to pipeline requests sent through a single instance of `SimpleRpcClient` until this restriction has been lifted. See also the overridable protected method `SimpleRpcClient.RetrieveReply`.

The basic pattern for using `SimpleRpcClient` is as follows:

```
using (IConnection conn = new ConnectionFactory()
                              .CreateConnection(args[0])) {
    using (IModel ch = conn.CreateModel()) {

        SimpleRpcClient client = new SimpleRpcClient(ch, /* ... */);
        // in the line above, the "..." indicates the parameters
        // used to specify the address to use to route messages
        // to the service.

        // The next three lines are optional:
        client.TimeoutMilliseconds = 5000; // defaults to infinity
        client.TimedOut += new EventHandler(TimedOutHandler);
        client.Disconnected += new EventHandler(DisconnectedHandler);

        byte[] replyMessageBytes = client.Call(requestMessageBytes);
        // other useful overloads of Call() and Cast() are
        // available. See the code documentation of SimpleRpcClient
        // for full details.
    }
}
```

Note that a single `SimpleRpcClient` instance can perform many (sequential) `Call()` and `Cast()` requests! It is recommended that a single `SimpleRpcClient` be reused for multiple service requests, so long as the requests are strictly sequential.

3.3. Event broadcasting

The event broadcasting pattern occurs when an application wishes to indicate a state change or other notification to a pool of applications without knowing precisely the addresses of each interested party. Applications interested in a certain subset of events use AMQP's exchanges and queue-bindings to configure which events are routed to their own private queues.

Generally, events will be broadcast through `topic` exchanges, although `direct` exchanges, while less flexible, can sometimes perform better for applications where their limited pattern-matching capability is sufficient.

3.3.1. Publishing events

To publish an event,

1. ensure the exchange exists

2. determine an appropriate routing key (for stocks, a key such as STOCK.IBM.NYSE might be appropriate; for other applications, other topic hierarchies will naturally arise. See the definition of the `topic` and `direct` exchange types in the AMQP specification)

3. publish the message

For example:

```
using (IConnection conn = new ConnectionFactory()
                             .CreateConnection(args[0])) {
    using (IModel ch = conn.CreateModel()) {

        IBasicProperties props = ch.CreateBasicProperties();
        FillInHeaders(props); // or similar
        byte[] body = ComputeBody(props); // or similar

        ch.BasicPublish("exchangeName",
                        "chosen.routing.key",
                        props,
                        body);
    }
}
```

See the documentation for the various overloads of `BasicPublish` on class `RabbitMQ.Client.IModel`.

3.3.2. Subscription

The class `RabbitMQ.Client.MessagePatterns.Subscription` implements most of the boilerplate of receiving messages (including, in particular, broadcast events) for you, including consumer declaration and management, but excluding queue and exchange declaration and queue binding. For example,

```
// "IModel ch" in scope.
Subscription sub = new Subscription(ch, "STOCK.IBM.#");
foreach (BasicDeliverEventArgs e in sub) {
  // handle the message contained in e ...
  // ... and finally acknowledge it
  sub.Ack(e);
}
```

will start a consumer on the queue using `IModel.BasicConsume`. It is assumed that the queue and any bindings have been previously declared.

`Subscription.Ack()` should be called for each received event, whether or not auto-acknowledgement mode is used, because `Subscription` internally knows whether an actual network message for acknowledgement is required, and will take care of it for you in an efficient way so long as `Ack()` is always called in your code.

For full details, please see the code documentation for the `Subscription` class.

3.3.3. Retrieving events - low-level approach

Usually, the high-level approach using `Subscription` is sufficient. Occasionally, however, there is a need to use the low-level AMQP primitives. The low-level approach to retrieving events is to bind a queue to the exchange concerned with an appropriate routing-key pattern specification. For instance, assuming that our application wanted to retrieve all prices regarding IBM on queue "MyApplicationQueue":

```
// "IModel ch" in scope.
ch.ExchangeDeclare("prices", "topic");
ch.QueueDeclare("MyApplicationQueue", false, true, true, null);
ch.QueueBind("MyApplicationQueue", "prices",
             "STOCK.IBM.#", false, null);
```

... followed by consumption of messages from "MyApplicationQueue" using `BasicGet` or `BasicConsume`. A more full example is given in the ApiOverview chapter.

3.3.4. Acknowledgment modes for event broadcasting

The same auto-acknowledgement/manual-acknowledgement decision as for point-to-point messaging is available for consumers of broadcast events, but the pattern of interaction introduces different tradeoffs:

- for high-volume messaging where it is occasionally acceptable to not receive one of the messages one is interested in, auto-acknowledgement mode makes sense

- for scenarios where every message matching our subscription needs to be delivered, manual-acknowledgement is appropriate

For more information, see the section on reliable message transfer below.

Note also that class `Subscription` takes care of acknowledgement and the various acknowledgement modes for you, so long as `Subscription.Ack()` is called for each received message.

3.4. Responsibility transfer

In cases where either

- a non durable exchange and/or queue is used[4], or

- a message is published with its `IBasicProperties.DeliveryMode` set to any value other than 2

the message is considered *transient*, and will not (usually) be written to disk as part of the interaction between the AMQP server and the various AMQP clients publishing and consuming messages.

When a message is transient, it means that the responsibility for ultimate processing of the message resides with the endpoints - the two AMQP clients involved: the publisher, and the consumer.

Responsibility can be partially transferred to the broker by using durable queues, durable exchanges, persistent-mode delivered messages (with `DeliveryMode` set equal to 2), and `Tx`-class transactions.

To transfer responsibility for delivery of a message to a broker

- ensure (ahead of time) that the target queue exists and is durable,

- select `Tx` mode using `IModel.TxSelect`,

- publish the message with the "mandatory" flag set and `DeliveryMode` set equal to 2, and

- commit the `Tx` transaction using `IModel.TxCommit`.

Once a broker replies with `CommitOk` (i.e. the `TxCommit()` call returns to the caller), it has taken responsibility for keeping the message on disk and on the target queue until some other application retrieves and acknowledges the message.

A commit is not required after every message: batching of publications may be done, depending on the precise delivery guarantees the publishing application requires.

Responsibility can also be placed with an external database, even further along the chain - see the section on interaction with external resources below.

3.5. Reliable message transfer

Messages can be transported between endpoints with different quality-of-service (QoS) levels. In general, failure cannot be completely ruled out, but it is important to understand the various delivery failure modes to understand the kinds of recovery from failure that are required, and the kinds of situation for which recovery is possible.

To reiterate: it is not possible to completely rule out failure. The best that can be done is to narrow the conditions in which failure can occur, and to notify a system operator when failure is detected.

3.5.1. At-least-once delivery

This QoS level assures that a message is delivered to its ultimate destination at least once. That is, a receiver may receive multiple copies of the message. If it is important that a side-effect only occur once for a given message, either at-most-once or exactly-once QoS levels should be used instead.

To implement at-least-once delivery:

- publish the message as usual, with some correlation identifier and reply-to address on it so that the receiver can acknowledge receipt to the sender
- when receiving a message, send an acknowledgement message back to the sender. If the message is an RPC request, then the RPC reply message is implicitly an acknowledgement of receipt of the request.

To reduce the need for resends of a message when publishing using an at-least-once QoS level, it can sometimes be appropriate to use `Tx`-class transactions, as described above in the section on responsibility transfer. Of course, if the `CommitOk` message from the server is lost, then a resend will have to be done, since the sending client doesn't know whether the server actually saw the message or not unless the `CommitOk` arrives.

Once of the important failure-modes when using at-least-once mode in conjunction with `Tx`-class transactions is the loss of atomicity that can arise when a completed transaction is published to the broker, but the whole transaction is only partially received on the consuming side before some kind of

failure occurs. In these situations, there will be partial duplication of a transaction: some of the messages in the transaction will be delivered more than once.

Deciding on a message-resend policy can be difficult. Some simple resend strategies are:

- resend if your connection is lost or some other crash occurs before you receive confirmation of receipt
- timeout and resend if you do not receive a confirmation within a few seconds. Make sure to double the timeout for each resend, to help avoid retry-related denial-of-service and network congestion.

3.5.2. At-most-once delivery

For at-most-once delivery, simply

- publish the message, once, as usual. No correlation identifier is required.
- receive the message in the consuming application, paying attention to the `Redelivered` flag on the delivery. The `Redelivered` flag will only be clear when the server believes that it is offering a message for consumption for the very first time. If any attempt at delivery has been made before, the `Redelivered` flag will be set.

The `Redelivered` flag is a very limited piece of information, giving only at-most-once QoS. In particular, it cannot be used for exactly-once QoS.

3.5.3. Exactly-once delivery

Exactly-once QoS is an extension of at-least-once QoS, with the addition of a barrier on the receiving side to ensure that a given message is never processed more than once, even if it is received more than once:

- publish the message as usual, with some correlation identifier and reply-to address on it so that the receiver can acknowledge receipt to the sender
- when receiving a message, keep a record of the messages that have been processed, and only process a received message if it has not been seen before.
- always send an acknowledgement message back to the sender, even if the message has been seen before.
- if the message is an RPC request, then the RPC reply message is implicitly an acknowledgement of receipt of the request. The RPC reply message *must* not change, even if it is sent a second time: it is important that even if one or more copies of the request or the reply message is lost or duplicated that the original requestor receives consistent information back from the requestee.

An interesting challenge in implementing the record of processed messages is deciding when it is acceptable to remove entries from the table (equivalent to being confident that no duplicates of those messages will appear in future). One option is to use the message expiry header in the

`IBasicProperties` structure, but be warned: versions of AMQP up to and including 0-9 do not specify the syntax or semantics of this field! Another option is to have a three-way exchange between the sender and recipient in which the sender promises not to send any further duplicates on the third leg of the interaction.

Still another option (perhaps the most elegant and space-efficient) is to use message identifiers that are strictly increasing as time goes by, so that a single "high water mark" can serve as a record of not only that message, but of all other messages preceding it, without taking up any additional space. Care must be taken when using this last option to account for request messages that arrive out-of-order or that go missing.

3.6. Coding with high-availability AMQP server pairs

In situations where continuous AMQP service is desired, the possibility of a server failure can be hedged against with some careful programming and the availability of a warm-standby AMQP server for failover.

The current version of the RabbitMQ .NET client library does not include any special support for failing over to a standby broker[5].

The main concerns when failing over are

• atomicity of published/acknowledged work units, and

• availability of configured resources on the backup server

Message producers should take care to use transactions in order to receive positive confirmation of receipt of a group of messages from a server, and should keep a record of the exchanges, queues and bindings they need to have available in order to perform their work, so that on failover, the appropriate resources can be declared before replaying the most recent transactions to recover.

Message consumers should be aware of the possibility of missing or duplicate messages when failing over: a publisher may decide to resend a transaction whose outcome is in doubt, or a transaction the publisher considered complete could disappear entirely due to failure of the primary AMQP server.

3.7. Interacting with external resources

A common pattern for an AMQP-connected service is to

1. receive a service request via an AMQP queue

2. update some external resource, such as a file or database

3. reply via AMQP, or at a minimum, acknowledge to the AMQP server that the message triggering the action has been completed

Often elements of the at-least-once or exactly-once patterns appear in conjunction with the external-resource pattern - specifically, the side-effects discussed in the section on reliable message transfer above are often effects on an external resource.

In cases where the exactly-once pattern appears in conjunction with an external resource, it's important to write code that is able at each step to determine whether the step has already been taken in some previous attempt at completing the whole transaction, and if it has, to be able to omit it in this attempt and proceed to the next step. For example:

- If a work-triggering request went missing, another copy will (eventually) arrive from the ultimate requestor.
- If the work was already performed, for instance a database table was updated, in a previous receipt of the work item in question, the service needs to keep a record of completion of the external work that is atomic with respect to the atomic work itself: for instance, within the same database transaction, some log of honoured requests could be updated, or the row being modified could be updated to include the ID of the request that caused the modification, as well as previous request-IDs that modified the row in question.

  This makes it important to be able to compress request IDs so that they do not take unbounded space in the log of performed work, and so that we do not need to introduce a full distributed garbage-collection protocol with the ultimate requestor. One way of doing this is to choose to use request IDs that are strictly increasing, so that a "high water mark" can be used; see also the section on exactly-once delivery above.

- Once the work is known to have been performed, and a reply has been produced (if there is one), the reply can be sent back to the requestor as many times as necessary. The requestor knows which replies it is expecting, and can discard unwanted duplicates. So long as duplicates of the same request always receive identical reply messages, the replier need not be careful about sending too many copies of the reply.
- Once the reply has been sent to the AMQP server, the request message can be acknowledged as received and processed with the AMQP server. In cases where there is no reply to a request, the acknowledgement is still useful to ensure that requests are not lost.
- Finally, if `Tx`-class transactions are in use, the work cycle can be closed off by invoking `IModel.TxCommit`, and the next request can be tackled.

4. ShutdownProtocols

4.1. General pattern of AMQP client shutdown

AMQP connection, model, channel and session objects share the same general approach to managing network failure, internal failure, and explicit local shutdown.

Each object has a lifecycle state, which can be one of

- *open*: the object is ready for use
- *closing*: the object has been explicitly shut down locally, has issued a shutdown request to any supporting lower-layer objects, and is waiting for the lower layer's shutdown-complete notification
- *closed*: the object has received a shutdown-complete notification from a lower layer, and has shut itself down in response.

Objects always end up in the closed state, without regard for whether the closure was triggered by a local application request, by an internal client library failure, or by a remote network request or network failure.

In general, objects possess the following shutdown-related members:

- a *shutdown event handler* `MulticastDelegate`, which will be fired when the object transitions to closed state
- a *close reason* property, to allow callers to inspect information about the circumstances that led to the object's shutdown
- a convenience property, useful for debugging, for testing whether the object is in an open state
- a *close* method that triggers the shutdown machinery for the object

4.2. Information about the circumstances of a shutdown

One class, `ShutdownEventArgs`, and one enumeration, `ShutdownInitiator`, are provided for applications to query objects about the circumstances of their shutdown.

4.2.1. ShutdownInitiator

`ShutdownInitiator` contains values representing the originator of a shutdown event. The three values are

- *Application*, for shutdown events that originated due to local application action (i.e. an explicit call to a `Close()` method)
- *Library*, for shutdowns caused by events originating within the RabbitMQ .NET client library itself, such as internal errors
- *Peer*, for shutdowns triggered by the peer, either through network failure or through explicit peer request (i.e. a received `channel.close` or `connection.close` method).

4.2.2. ShutdownEventArgs

The `ShutdownEventArgs` class is used to carry the full circumstances of a shutdown, including a `ShutdownInitiator` value and any relevant reply code and/or text, AMQP class and/or method ID, and possible "cause" object (usually an `Exception`).

A `ShutdownEventArgs` instance is carried as the argument to various shutdown event handler delegates used in the system, and one is also carried by `AlreadyClosedException` and `OperationInterruptedException`.

4.3. The CloseReason properties, and idempotency of Close() calls

Closable AMQP objects generally have a `CloseReason` property, of type `ShutdownEventArgs`, which can be used to determine the situation in which the object was shut down. A null value for the property indicates that no close reason is available, because the object is not yet shut down. Once the close reason is set for an object, it will not change.

A `Close()` method on an object, with the exception of `IConnection`, can be called several times without penalty. Since a close reason can only be set once per object, calls after the first are effectively ignored.

Calls to `Close()` will not return to the caller until the full shutdown procedure is completed for the object concerned.

4.4. Atomicity, and use of the IsOpen flags

Use of the various `IsOpen` properties is not recommended for production code, since the property may change value between the time that you look at it and the time you use the object. For example, the following code contains a race condition:

```
public void BrokenMethod(IModel model) {
  // RACE CONDITION
  if (model.IsOpen) {
    // The following method call depends on the model being in open state.
    // The state may change between the IsOpen call and the TxCommit call!
    model.TxCommit();
  }
}
```

The pattern that should be used is to ignore the *IsOpen* property, and instead simply attempt the action desired. An `OperationInterruptedException` will be thrown in cases where the object is in an invalid state:

```
public void BetterMethod(IModel model) {
  try {
    ...
    model.TxCommit();
  } catch (OperationInterruptedException ope) {
    // Handle one of the many possible reasons why the commit or any
    // other operation before it couldn't be handled
  } catch (IOException ioe) {
    // Check why socket was closed
  }
}
```

In the above pattern we also catch `IOException`s caused by a `SocketException`. Such exceptions may be raised when the broker closes the connection unexpectedly.

4.5. Specific shutdown protocols

4.5.1. IModel

In the current version of the RabbitMQ .NET client library, supporting AMQP specification versions 0-8 and 0-9, IModel encapsulates both a session and a channel layered on top of a connection. Shutting down an IModel instance causes the underlying session and channel to be closed, but does not close the underlying IConnection.

```
delegate ModelShutdownEventHandler(IModel model, ShutdownEventArgs reason);
event ModelShutdownEventHandler ModelShutdown;
ShutdownEventArgs CloseReason { get; }
bool IsOpen { get; }
void Close(ushort replyCode, string replyText);
```

Any time an AMQP operation is invoked on an `IModel` that is not in the opened state, an `AlreadyClosedException` is thrown.

4.5.2. IConnection

IConnection encapsulates an AMQP connection. When the connection shuts down, either through explicit user action or through network failure or similar, all associated channels, sessions, and models are shut down. Connection can be gracefully closed using `Close()` or forced to close using `Abort()` methods. `Close(int timeout)` and `Abort(int timoeut)` methods allow for specifiyng timeout for underlying connection close operations to finish. If such timeout is reached we force the socket to close and exit. Default value is `Timeout.Infinite`. A call on `Close()` method on `IConnection` object

which was previously closed, will throw `AlreadyClosedException` and new close reason will be ignored. `Abort()` can be called several times, since any attempt to abort closed connection will be ignored. `ShutdownReport` property returns any errors that occured during connection close operations as a list of `ShutdownReportEntry` objects.

```
delegate ConnectionShutdownEventHandler(IConnection connection,
                                        ShutdownEventArgs reason);
event ConnectionShutdownEventHandler ConnectionShutdown;
ShutdownEventArgs CloseReason { get; }
bool IsOpen { get; }
void Close();
void Close(int timeout);
void Abort();
void Abort(int timeout);
IList ShutdownReport { get; }
```

4.5.3. ISession

Sessions are currently an internal, implementation-only concept, and *should not be used by applications*, as they are not part of the public API.

```
delegate SessionShutdownEventHandler(ISession session,
                                     ShutdownEventArgs reason);
event SessionShutdownEventHandler SessionShutdown;
ShutdownEventArgs CloseReason { get; }
bool IsOpen { get; }
void Close(ShutdownEventArgs reason);
```

5. TroubleShooting

5.1. Heartbeat issue with Mono 1.1.17.1 on Redhat FC6/x86-64

Fix: Upgrade Mono to 1.2.x.

Symptom: Exception while connecting. Appears immediately, not after the expected 3 or 6 seconds.

Example symptom trace:

```
Unexpected connection Connection(d49c7727-3d56-4101-909e-5a7ff17b5647,
amqp-0-9://127.0.0.1:5672) closure: AMQP close-reason, initiated by Library,
code=0, text="End of stream", classId=0, methodId=0,
cause=System.IO.EndOfStreamException: Heartbeat missing with
```

```
heartbeat == 3 seconds
at RabbitMQ.Client.Impl.ConnectionBase.HandleSocketTimeout () [0x00000]
at RabbitMQ.Client.Impl.ConnectionBase.MainLoopIteration () [0x00000]
at RabbitMQ.Client.Impl.ConnectionBase.MainLoop () [0x00000]
```

Problematic Mono version:

```
$ mono --version
Mono JIT compiler version 1.1.17.1, (C) 2002-2006 Novell, Inc and Contributors.
        www.mono-project.com
        TLS: __thread
        GC: Included Boehm (with typed GC)
        SIGSEGV: normal
        Disabled: none
```

6. Examples

This section gives an overview of the simple examples provided with .Net client distribution.

6.1. How to run examples

From the command line go to the directory where RabbitMQ .Net/C# client is installed, and then to the bin directory. All of the examples require additional arguments to run. Typing the name of the example, e.g. `SendString.exe` will print the list of required and optional arguments to the program.

6.2. Examples overview

6.2.1. AddClient

Example that illustrates the usage of remote procedure calls. The client sends a list of numbers (given in the program's arguments) using the 'AddServer' routing key. The message is then read by the client which performs the function of the server, namely `AddServer`. The sum is calculated and the result is sent back to AddClient via an acknowledgement.

6.2.2. AddServer

`AddServer` performs the function of the remote procedure call server. On the lower level it uses the Subscription pattern that reads messages from the queue, calculates the sum of the numbers included in the body of the message and sends an acknowledgement with the result.

6.2.3. DeclareQueue

This example creates a queue (if it does not exist yet) and binds it to multiple exchanges using the exchange name and routing key tuple.

6.2.4. ExceptionTest

`ExceptionTest` allows to see the usage of Shutdown protocols in a real scenario. We attach three event handlers to both the connection and the model, which then print information messages in an order defined in the shutdown protocol.

6.2.5. LogTail

The `LogTail` example makes use of the `Subscription` pattern described in the MessagingPatterns section. When the exchange name is empty, it declares a named queue, otherwise it declares an exchange of the specified type and a temporary queue from which it will consume messages. `LogTail` quietly consumes messages until it receives a message with only 'quit' in the body.

6.2.6. SendMap

`SendMap` publishes a single message to the exchange which is constructed from key/value pairs given as arguments. Keys that start with '+' are placed in the body of the message, whereas those starting with '-' go to the header. Supported values' types are: string/byte array, byte array (base-64), boolean, 32-bit integer, double precision float and fixed-point decimal. Refer to the info printed in the command line for detailed information on how to specify values with particular types.

6.2.7. SendString

The SendString example performs a single message publish, according to the given routing key. If the exchange name is not empty it declares an exchange with the given name and type. Otherwise it declares a queue with the name given in the routing key.

6.2.8. SingleGet

SingleGet will try to retrieve a message from the queue, the name of which is given in the command line. If there is no message in the queue it will close the connection and exit.

7. BuildingTheClient

7.1. Source Tree Layout

The source tree is laid out as follows:

- `docs` - Contains the AMQP specification XML files, used to generate code in `projects/client/Apigen` project

- `lib` - Third-party libraries and programs referenced by the code or used during the build

- `projects` - Source code to the various parts of the system

  - `client` - Source code and project files for building the client

    - `Apigen` - Project for the XML specification parser and code generator application

    - `ApigenBootstrap` - Project for creating a library out of some parts of RabbitMQ.Client's source code that are used in Apigen

    - `RabbitMQ.Client` - The client library project

    - `Unit` - Unit tests for the client library

  - `wcf` - WCF binding source code and project files

    - `RabbitMQ.ServiceModel` - The WCF binding project

  - `examples` - Example projects

    - `client` - Example code and projects for using the client library

    - `wcf` - Example code and projects for using the WCF binding

- `Local.props` - Local customizations for the build process. Copy the contents of `Local.props.example` in this file, to start with

- `RabbitMQDotNetClient.sln` - The Visual Studio solution file which contains all the projects

7.2. Build Prerequisites

To compile the RabbitMQ .NET client library, you will need

- a .NET development environment:

  - Microsoft .NET 2.0 (or later)

  - Microsoft .NET 3.0 (or later) if you want to build the WCF binding too

- Microsoft Visual Studio 2005 or later (optional)

To run the unit tests, you will need

- NUnit 2.5 or later; included in the `lib` directory

7.3. Configuring Your Tree

Copy `Local.props.example` to `Local.props`, and edit it if necessary (the properties in this file have suggestive names). Set `PropTargetFramework` to `v1.1`, `v2.0`, `v3.0` or `v3.5`. If you would like to build under Mono, only `v1.1` and `v2.0` are supported; also set `PropUsingMono` to `true` in this case.

7.4. Building

7.4.1. On Windows, with Visual Studio

Simply open `RabbitMQDotNetClient.sln` with Visual Studio and build.

If you are using a version of Visual Studio later than Visual Studio 2005, you will have to convert the solution to make it compatible with your version of Visual Studio. This process is automated and you are presented with a wizard when you first open the solution file.

The projects in the RabbitMQ .NET client contain some customizations in order to allow generating source files and running NUnit tests as part of the build process. You might be presented with warnings claiming that the projects might be unsafe. You will have to instruct Visual Studio to open all projects normally.

To skip running the NUnit tests as part of the build, select the `DebugNoTest` build configuration.

Note: If you change a property in `Local.props` while the solution is loaded in Visual Studio, you will have to close the solution and reopen it, for the changes to take effect.

7.4.2. On Windows, without Visual Studio

To build the client library without Visual Studio, first you need to make sure you have the .NET's framework directory in your `PATH` environment variable. The .NET framework directory can usually be found at `%WINDIR%\Microsoft.NET\Framework\v2.0.50727` for .NET 2.0 or `%WINDIR%\Microsoft.NET\Framework\v3.0` for .NET 3.0.

Open the Command Prompt (Start -> Run -> cmd) or Cygwin (if you have it installed) and change to the client's directory. Now type `msbuild.exe` and hit return to start building the solution.

Refer to MSBuild's documentation for information on valid command line switches for `msbuild.exe`.

7.4.3. On Linux, using Mono

For building under Mono, please refer to
http://www.rabbitmq.com/build-dotnet-client.html#building-on-linux.

7.5. Building the WCF binding

Building the WCF binding is enabled by default in `Local.props` if the target framework is `v3.0` or later (only on Windows).

To disable building the WCF binding, set `PropBuildWcf` to `false`.

Building the WCF binding under Mono is currently unsupported due to limitations of Mono.

7.6. Build Products

The build products available after a successful build are:

- `projects/client/RabbitMQ.Client/build/bin/RabbitMQ.Client.dll` - The final client assembly
- Example client programs, like
  `projects/examples/client/SendString/build/bin/SendString.exe` and
  `projects/examples/client/LogTail/build/bin/LogTail.exe`

If building the WCF binding is enabled, there will also be:

- `projects/wcf/RabbitMQ.ServiceModel/build/bin/RabbitMQ.ServiceModel.dll` - The WCF binding library
- Example programs for the WCF binding, like
  `projects/examples/wcf/Test/build/bin/RabbitMQ.ServiceModel.Test.exe`

Other build products:

- `gensrc/RabbitMQ.Client/autogenerated-api-*.cs` - Protocol-specific autogenerated code, produced by Apigen

- `projects/client/RabbitMQ.Client/build/bin/RabbitMQ.Client.xml` - `csc`-generated XML documentation of the client assembly

- `projects/wcf/RabbitMQ.ServiceModel/build/bin/RabbitMQ.ServiceModel.xml` - `csc`-generated XML documentation of the WCF binding

8. ImplementationGuide

The internals of the RabbitMQ .NET AMQP client library implementation live in `src/client/impl`, in C# namespace `RabbitMQ.Client.Impl`. Most of the classes mentioned below are in this namespace, modulo conspicuous exceptions such as `IModel` and `IProtocol`, which are instead in namespace `RabbitMQ.Client`.

8.1. Connections, Channels, Sessions and Models

Each AMQP connection is represented by an instance of `ConnectionBase`. Each connection has one thread running in the background, which reads and dispatches frames from the connection socket, as well as acting as a serialization point for outbound AMQP commands. There may also be two additional threads responsible for `Heartbeat` support (In and Outbound), if it was negotiated during the connection open process.

Each `ConnectionBase` instance has an instance of `SessionManager`, which is responsible for allocation of channel numbers, and maintenance of the map from channel numbers to `ISession` instances.

Each AMQP channel, that is, an uninterpreted, independent stream of commands, is represented by an instance of `ISession`. There are three implementations of this interface at the time of writing: `Session`, which provides the normal operation of a channel, `QuiescingSession`, which represents a channel in the process of an abnormal shutdown, and `MainSession` which controls special channel zero responsible for sending/receiving frames and used in `ConnectionBase`. The `ISession` instances do not interpret the commands that travel across them.

Interpretation of AMQP commands sent and received is done by the "model" code, implemented for the most part in `ModelBase`. This is the class that contains the majority of the implementation of the public AMQP API from `IModel`, as well as the private AMQP operations from `IFullModel`. It is also `ModelBase` that provides the request/response correlation during an AMQP RPC-style synchronous operation, in conjunction with `RpcContinuationQueue`.

8.2. Channel zero

Channel zero is a special connection-management channel, and is implemented using field `ConnectionBase.m_model0`. See also field `ModelBase.m_connectionStartCell` and methods `ModelBase.HandleConnection`*Methodname*, all of which are an awkward part of `ModelBase` only relevant to operations on channel zero. `MainSession` implements the special behaviour of the channel zero.

8.3. Information dataflow from the server to the client
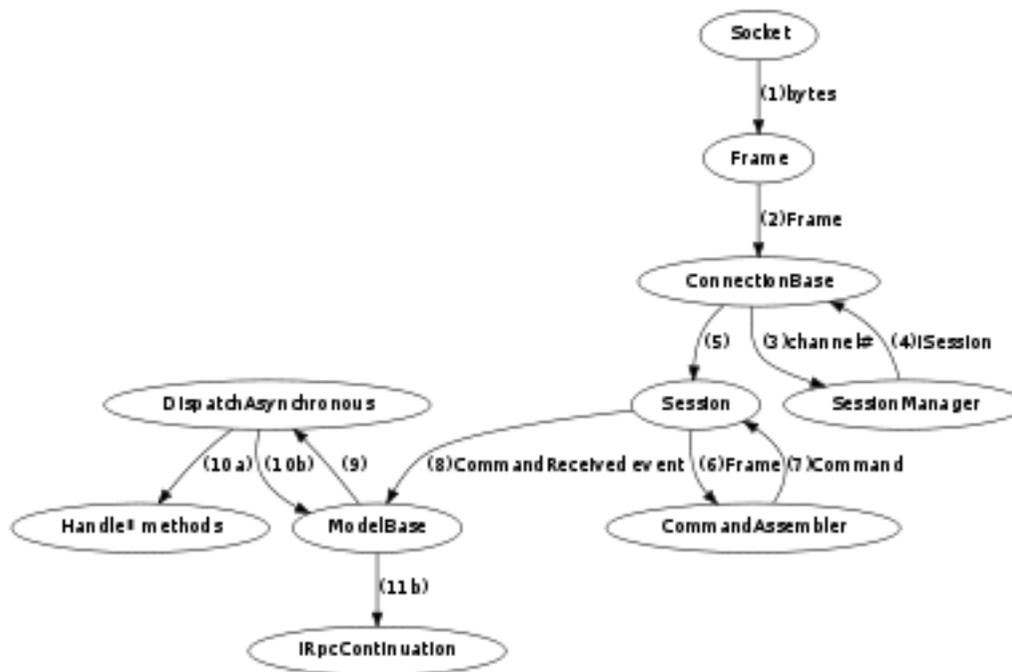
AMQP commands transmitted from the server to the client are processed by each `ConnectionBase`'s connection thread, which runs `ConnectionBase.MainLoop` until the connection is terminated.

`MainLoop` reads a frame from the connection's socket, blocking until either some traffic arrives or the socket's read-timeout exception fires. A timeout in .Net means that the socket is in an unusable state, hence the whole `Connection` needs to be closed. If `Heartbeat` support is active then two additional loops run in separate threads: `ConnectionBase.HeartbeatReadLoop` and `ConnectionBase.HeartbeatWriteLoop`.

`HeartbeatReadLoop` continuously checks if our connection was inactive for too long and we missed too many `Heartbeat` frames the broker. In such case `EndOfStreamException` occurs and the connection is closed.

`HeartbeatWriteLoop` continously checks the activity of the connection and forces `Heartbeat` frame to be sent to the broker if it was idle for too long.

Once a complete frame has been read from the socket, it is dispatched with a call to `ISession.HandleFrame` to the `ISession` object responsible for the channel number carried within the frame.

In the normal case, the `ISession` implementation will be `Session`, whose `HandleFrame` method makes use of an instance of `CommandAssembler` to stitch sequentially-received frames together into whole AMQP commands. As soon as a complete command is available, it is sent through the `CommandReceived` event handler on `Session`, which in the usual case will result in an invocation of `ModelBase.HandleCommand`.

`ModelBase.HandleCommand` uses `DispatchAsynchronous`, which is part of the automatically-generated code produced by Apigen, to decide whether the received command is an asynchronous event (including RPC responses that have been marked in `IModel` as `AmqpForceOneWay` etc) or an RPC response that should release a waiting `IRpcContinuation`, about which more below.

8.4. Information dataflow from the client to the server

There are two styles of interaction available to `ModelBase`:

• synchronous, RPC-style interaction, e.g. `ExchangeDeclare`, `BasicGet` and `BasicConsume`.

• asynchronous, event-style interaction, e.g. `BasicAck`, `BasicPublish`.

Sometimes, what would normally be an RPC-style interaction must, for implementation reasons, instead be treated as two loosely associated asynchronous messages, for instance when `nowait=true` on methods like `ExchangeDeclare`, or in the case of methods like `BasicConsume`. For more information, see the section on `AmqpForceOneWay` in the section on ApiGen.

The `ModelBase` implementation does not directly choose which alternative to use: instead, the autogenerated code makes calls to `ModelBase.ModelRpc` and `ModelBase.ModelSend` as directed by the annotations on `IFullModel` (again, see the section on ApiGen for more details).

`ModelSend` is the simpler of the two, as no record of an awaited reply is required: the command is simply formatted and sent down the socket (via `Command.Transmit`, via `SessionBase.Transmit`).

`ModelRpc` is more involved. Calls to `IModel` methods that (indirectly) invoke `ModelRpc` should only occur in the application's thread, and should never occur in the connection's thread (see also the section on threading issues and deadlock in the ApiOverview).

8.5. Client-to-server AMQP RPC

When an application thread calls an `IModel` method that in turn calls `ModelBase.ModelRpc`, the application thread is suspended, the request is sent to the server, and when the response arrives, the suspended application thread is awakened. The interface between the application thread and the connection thread is usually an instance of `SimpleBlockingRpcContinuation`, the methods of which are called from the connection thread when a reply or exceptional condition arrives.

If `SimpleBlockingRpcContinuation.HandleCommand` is called, the reply `Command` is handed back to `ModelRpc`, which then hands it back through the autogenerated code to the application. Otherwise, `SimpleBlockingRpcContinuation.HandleModelShutdown` will be called due to some error condition on the channel, in which case `OperationInterruptedException` will be thrown from `SimpleBlockingRpcContinuation.GetReply`, through `ModelRpc`, and out to the application.
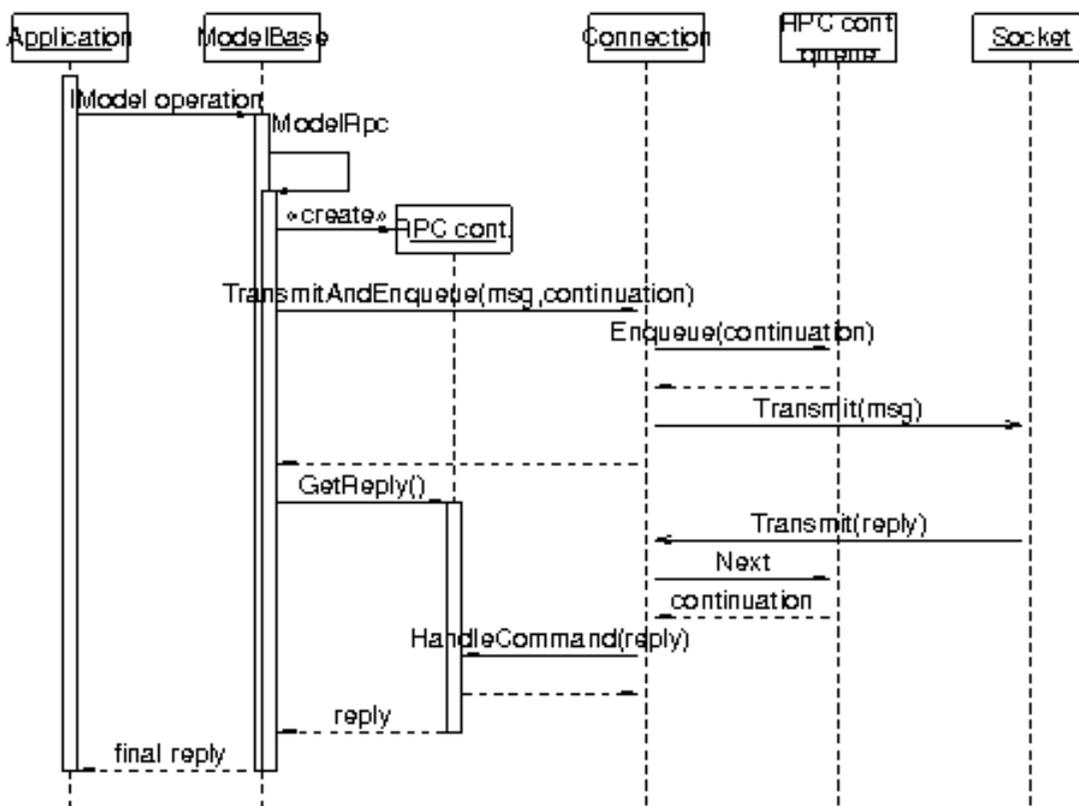
8.6. Constants from the specification

From numerous places in the implementation source code:

```
// We use spec version 0-9 for common constants such as frame types,
// error codes, and the frame end byte, since they don't vary *within
```

```
// the versions we support*. Obviously we may need to revisit this if
// that ever changes.
using CommonFraming = RabbitMQ.Client.Framing.v0_9;
```

If this needs revisiting, a good place to start would be extending `IProtocol` to cover the variation.

9. ApiGen

Apigen is the program that reads the AMQP XML specification documents, and autogenerates the code found after a build in the `build/gensrc/autogenerated-api-*.cs` files.

It is automatically invoked as part of the NAnt `default.build` script.

9.1. Compilation Process Overview
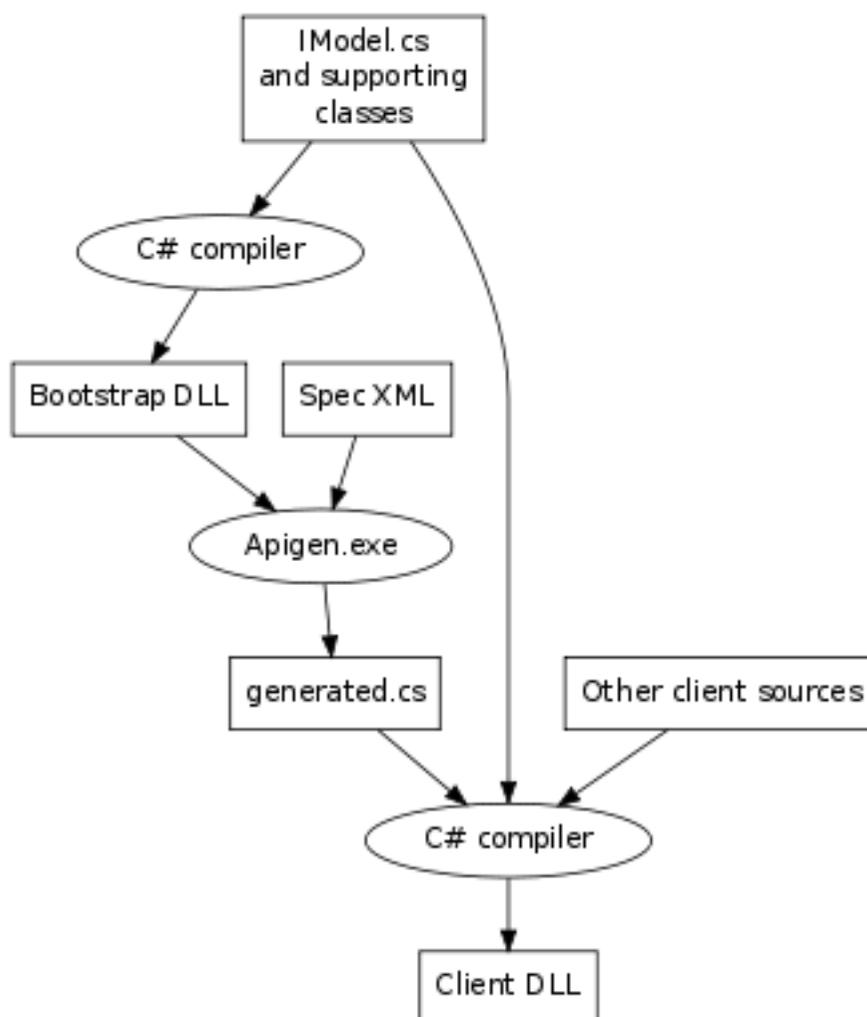
Apigen uses the `IModel` interface definition to guide its code generation process in conjunction with the particular XML specification file chosen.

First, `IModel` and a minimal number of its supporting classes and interfaces are compiled together using the C# compiler, resulting in `build/bin/apigen-bootstrap.dll`.

Apigen then uses .NET reflection while reading the XML specification file (from `docs/specs/amqp*.xml`), and finally generates the output C# code.

The generated code is then compiled with all the C# API and implementation classes, resulting in `build/bin/RabbitMQ.Client.dll`.

9.2. Invoking Apigen

```
Usage: Apigen.exe [options ...] <input-spec-xml> <output-csharp-file>
  Options include:
    /apiName:<identifier>
    /n:<name.space.prefix>
    /v:<majorversion>-<minorversion>
  The apiName option is required.
```

The `<input-spec-xml>` parameter should be the path to the AMQP specification file to digest. The `<output-csharp-file>` should be the location of the output file to generate.

The available options are:

- `apiName:<identifier>` - Specifies the identifier that will be used to represent the generated `IProtocol` in the `RabbitMQ.Client.Protocols` class.

- `n:<name.space.prefix>` - Supplying a namespace prefix of, for instance, `foo`, will cause the generated code to be placed in `RabbitMQ.Client.Framing.foo` and `RabbitMQ.Client.Framing.Impl.foo`. If this parameter is not supplied, a default will be constructed from the version (either the `/v:` command-line option, or the version specified in the spec XML): if the version is N-M, for instance 0-8, then the effect is as if `/n:vN_M` was supplied (e.g. `/n:v0_8`).

- `v:<majorversion>-<minorversion>` - Overrides the version numbers found in the spec XML file. Used as part of the namespaces for generated code if no `/n:` option is supplied.

9.3. Generated Classes
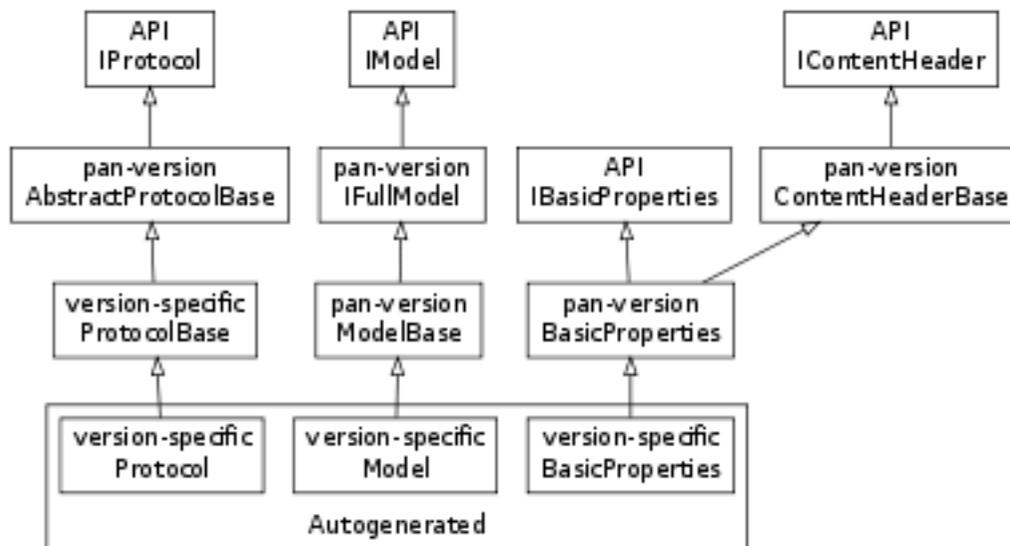
Generated classes are placed in namespaces

- `RabbitMQ.Client.Framing.`*VersionSpecificPart*
- `RabbitMQ.Client.Framing.Impl.`*VersionSpecificPart*

The generated classes include, in the semi-public namespace:

- class `RabbitMQ.Client.Framing.`*VersionSpecificPart*`.Protocol`
- class `RabbitMQ.Client.Framing.`*VersionSpecificPart*`.Constants`
- interface `RabbitMQ.Client.Framing.`*VersionSpecificPart*`.I`*ClassnameMethodname*, one per AMQP XML specification method
- class `RabbitMQ.Client.Framing.`*VersionSpecificPart.Classname*`Properties`, one per AMQP XML content-bearing class

and in the private/implementation-specific namespace:

- class `RabbitMQ.Client.Framing.Impl.`*VersionSpecificPart*`.Model`
- class `RabbitMQ.Client.Framing.Impl.`*VersionSpecificPart.ClassnameMethodname*, one per AMQP XML specification method

The `Protocol` class implements `IProtocol`. Its most important generated methods are `DecodeMethodFrom` and `DecodeContentHeaderFrom`.

The `Constants` class collects the various numeric constants defined in the XML specification file.

Each `I`*ClassnameMethodname* interface represents a protocol-specific variant of a single AMQP method, and its corresponding `Impl`-namespaced *ClassnameMethodname* class is the implementation of the interface.

Each *Classname*`Properties` class is a protocol-specific implementation of a common content header interface.

The `Model` class implements the autogenerated methods from the `IFullModel` definition (about which much more below, in the "The Common Model" section), as well as implementing the `DispatchAsynchronous` method used to decide if an incoming command is an asynchronous event or a reply to a synchronous RPC request.

The inheritance relationships are as follows (for specification version 0-9; other versions follow, *mutatis mutandis*):

for `Protocol`:

- `RabbitMQ.Client.Framing.v0_9.Protocol`

- is autogenerated

- is protocol-specific

- extends `RabbitMQ.Client.Framing.Impl.v0_9.ProtocolBase`

- `RabbitMQ.Client.Framing.Impl.v0_9.ProtocolBase`

  - is written by hand

  - is very short

  - is protocol-specific

  - is implemented in file `src/client/impl/v0_9/ProtocolBase.cs`

  - extends `RabbitMQ.Client.Impl.AbstractProtocolBase`

- `RabbitMQ.Client.Impl.AbstractProtocolBase`

  - is written by hand

  - is protocol-neutral

  - implements `RabbitMQ.Client.IProtocol`

- `RabbitMQ.Client.IProtocol`

  - is protocol-neutral

  - is public API.

for `Model`:

- `RabbitMQ.Client.Framing.Impl.v0_9.Model`

  - is autogenerated

  - is protocol-specific

  - extends `RabbitMQ.Client.Impl.ModelBase`

- `RabbitMQ.Client.Impl.ModelBase`

  - is written by hand

  - is long and complex

  - is protocol-neutral

  - is implemented in file `src/client/impl/ModelBase.cs`

  - implements `RabbitMQ.Client.Impl.IFullModel`

- `RabbitMQ.Client.Impl.IFullModel`

- is written by hand

- is protocol-neutral

- is compiled in to Apigen and used to guide the compilation process

- extends `RabbitMQ.Client.IModel`

- `RabbitMQ.Client.IModel`

  - is written by hand

  - is protocol-neutral

  - contains the main public API for performing AMQP operations.

for `BasicProperties`:

- `RabbitMQ.Client.Framing.v0_9.BasicProperties`

  - is autogenerated

  - is protocol- and content-class-specific

  - extends `RabbitMQ.Client.Impl.BasicProperties`

- `RabbitMQ.Client.Impl.BasicProperties`

  - is written by hand

  - is protocol-neutral, but content-class-specific

  - extends `RabbitMQ.Client.Impl.ContentHeaderBase`

  - implements `RabbitMQ.Client.IBasicProperties`

- `RabbitMQ.Client.Impl.ContentHeaderBase`

  - is written by hand

  - is protocol- and content-class-neutral

  - implements `RabbitMQ.Client.IContentHeader`

- `RabbitMQ.Client.IBasicProperties`

  - is written by hand

  - is protocol-neutral, but content-class-specific

  - is public API.

9.4. The Common Model

In order to accommodate differences in broker implementations, we supply a common interface that all supported protocol variants must implement. The common interface is in two pieces:

• `RabbitMQ.Client.IModel` - the public API

• `RabbitMQ.Client.Impl.IFullModel` - internal, implementation-specific pieces of the protocol; extends the `IModel` interface.

Apigen is linked with these interfaces, and at runtime uses reflection to look for the `IFullModel` interface (see definition of `Apigen.m_modelType`).

The definitions in `IFullModel`, taken together with the class and method definitions in the AMQP specification XML, guide the code generation process.

9.4.1. The main model interface

The interface itself contains both

• members that should be autogenerated, and

• members that must not be autogenerated, that will be supplied by hand-written code.

9.4.2. Content header interfaces

```
[AmqpContentHeaderFactory("basic")]
IBasicProperties CreateBasicProperties();

[AmqpContentHeaderFactory("file")]
IFileProperties CreateFileProperties();

// etc.
```

Use the `AmqpContentHeaderFactory` attribute to specify that a particular method is to be implemented by constructing an instance of a protocol-specific implementation of a common, protocol-neutral interface.

The parameter to the attribute specifies the content class that should be used. It may not be null.

9.4.3. Mapping methods to RPC requests

The name of the AMQP method to associate with a given `IFullModel` method is derived by analysing the name of the C# method:

```
ExchangeDeclare --> (split by CamelCasing)
Exchange Declare --> (lookup each piece in the XML file)
class=exchange, method=declare
```

9.4.3.1. AmqpMethodMapping

If the name of the method does not match any AMQP name, a mapping must be explicitly provided:

```
[AmqpMethodMapping(null, "channel", "open")]
void _Private_ChannelOpen(string outOfBand);
```

The first parameter to the attribute specifies the specification version to which the attribute should apply, or null if it should apply to any and all specification versions.

The second and third parameters are the AMQP specification class and method names, respectively, to use when building a mapping for this C# method.

9.4.3.2. AmqpMethodDoNotImplement

The `AmqpMethodDoNotImplement` attribute can be used on any method to cause code-generation to skip that method.

```
[AmqpMethodDoNotImplement(null)]
void ExchangeDeclare(string exchange, string type);
```

It should be used to mark interface methods that are backed by hand-written code: for example, the `ExchangeDeclare` overload above is a convenience method, backed by hand-written code that invokes a different overload of `ExchangeDeclare` with all the missing parameters filled in with sensible defaults.

Another case in which hand-written code is required is when non-standard RPC processing is needed, for instance in the cases of `BasicConsume` and `BasicCancel`. These delegate (in hand-written code) to the automatically-generated `_Private_BasicConsume` and `_Private_BasicCancel` methods. This pattern provides an opportunity to combine automatically-generated marshalling code with other concerns not related to the framing or transmission of a request.

The parameter to the attribute specifies the specification version to which the attribute should apply, or null if it should apply to any and all specification versions.

9.4.4. Mapping method parameters to RPC request fields

Parameters in a mapped method call are matched to the corresponding fields in an AMQP command by analysing the parameter name:

```
prefetchCount --> (split on CamelCasing)
"prefetch count"
<-- "prefetch count" (in AMQP specification 0-8)
<-- "prefetch-count" (in AMQP specification 0-9)
```

9.4.4.1. AmqpFieldMapping

If the name of the parameter to a method call does not fit such an analysis, the `AmqpFieldMapping` attribute should be used on the parameter concerned:

```
void _Private_BasicConsume(...,
                           bool exclusive,
                           bool nowait,
                           [AmqpFieldMapping("RabbitMQ.Client.Framing.v0_9",
                                             "filter")]
                           IDictionary arguments);
```

The parameter to the attribute specifies the specification version to which the attribute should apply, or null if it should apply to any and all specification versions.

The second parameter is the AMQP XML field name to match to the parameter.

9.4.4.2. AmqpNowaitArgument

Some AMQP command requests have a "nowait" boolean field, which affects the interaction style of the command, turning it from a synchronous RPC request into an asynchronous command request. In order to generate the correct workflow code, Apigen needs to know which (if any) of a method's parameters will change the server's interpretation of the interaction style of the command. The `AmqpNowaitArgument` attribute should be used for this purpose:

```
void _Private_ExchangeDelete(string exchange,
                             bool ifUnused,
                             [AmqpNowaitArgument(null)]
                             bool nowait);
```

When `AmqpNowaitArgument` decorates a particular parameter, if that parameter is true at runtime, the generated code will send the request asynchronously; otherwise it will send it synchronously.

It is an error to apply `AmqpNowaitArgument` to methods mapped to commands that are not synchronous.

The first parameter to the attribute specifies the specification version to which the attribute should apply, or null if it should apply to any and all specification versions.

The second, optional, parameter to the attribute specifies the value to return from the method in cases where the server's reply is not waited for. It can be safely omitted for methods returning void, or methods returning a reference type (in which case null is returned). For other methods, it should be a string containing a C# code fragment to place in the generated code.

9.4.4.3. AmqpContentHeaderMapping and AmqpContentBodyMapping

Some methods take AMQP content, in the form of subsequent frames to be sent on the wire: content headers followed by content body. The `AmqpContentHeaderMapping` and `AmqpContentBodyMapping` attributes mark the parameters of a method that should be sent as the content header and content body frames, respectively:

```
void _Private_BasicPublish(ushort ticket,
                           string exchange,
                           string routingKey,
                           bool mandatory,
                           bool immediate,
                             [AmqpContentHeaderMapping]
                           IBasicProperties basicProperties,
                             [AmqpContentBodyMapping]
                           byte[] body);
```

Parameters so marked will not be mapped to fields in the command request.

9.4.5. Mapping RPC responses to return values

Commands mapped to methods on `IFullModel` may have a response command that contains useful information that should be exposed as the return value of the C# method. If the C# method returns void, no information will be returned, but the method will not return until the response has arrived from the server.

9.4.5.1. AmqpMethodMapping

If `AmqpMethodMapping` is applied to the return value of a mapped C# method, it causes the generation of code that will wait for the named reply method instead of whatever default could be gleaned from reading the specification XML:

```
[return: AmqpMethodMapping(null, "connection", "tune")]
ConnectionTuneDetails ConnectionStartOk(IDictionary clientProperties,
                                        string mechanism,
                                        byte[] response,
```

```
                                  string locale);
```

In the example above, the XML specification talks about request-reply pairs involving `connection.start` and `connection.start-ok`, and `connection.tune` and `connection.tune-ok`. Because these are seen from the server's perspective, and we are implementing a client, we need to talk about a request-reply pair involving `connection.start-ok` and `connection.tune` instead.

9.4.5.2. AmqpFieldMapping

If `AmqpFieldMapping` is applied to the return value of a mapped C# method, it causes a single field to be extracted from the reply command and returned.

If it is not applied, however, then if the method does not return void, the return type is assumed to be valuetype (a.k.a. struct) with annotations specifying how to map AMQP reply method fields onto fields of the returned struct.

```
[return: AmqpFieldMapping(null, "messageCount")]
uint QueuePurge(...);
```

The above example extracts and returns just the "message-count" field of the `queue.purge-ok` reply.

```
[return: AmqpMethodMapping(null, "connection", "tune")]
ConnectionTuneDetails ConnectionStartOk(IDictionary clientProperties,
                                        string mechanism,
                                        byte[] response,
                                        string locale);

public struct ConnectionTuneDetails
{
    public ushort m_channelMax;
    public uint m_frameMax;
    public ushort m_heartbeat;
}
```

The above example extracts the "channel-max", "frame-max" and "heartbeat" fields of the reply, and inserts them into a fresh `ConnectionTuneDetails` structure. The definitions of the fields may have `AmqpFieldMapping` attributes applied to them if their names do not match the names given in the specification XML.

9.4.5.3. AmqpForceOneWay

In some cases where a method mapping would otherwise be generated as a synchronous RPC, we need to send it instead as an asynchronous command, decoupling the request from the response and permitting the weaving of non-framing-related concerns into the workflow. In these cases, applying

`AmqpForceOneWay` to the method causes it to always be sent as an asynchronous request. Methods so marked must return void.

```
[AmqpForceOneWay]
[AmqpMethodMapping(null, "basic", "get")]
void _Private_BasicGet(ushort ticket,
                       string queue,
                       bool noAck);

void HandleBasicGetOk(...);

void HandleBasicGetEmpty();
```

The example above has been marked AmqpForceOneWay because the specification XML supplies two possible responses, `basic.get-ok` and `basic.get-empty`, which does not fit the simple RPC request-reply pattern implemented for many of the other commands.

Since `basic.get` will always result in one or the other of the mentioned result methods, it's important to handle these methods as if they were asynchronous events from the broker, matching up the request and response in hand-written code.

9.4.6. Mapping asynchronous events

Methods which have a name starting with the string "Handle" are assumed by Apigen to be event handlers for responding to asynchronous events.

```
void HandleBasicDeliver(string consumerTag,
                        ulong deliveryTag,
                        bool redelivered,
                        string exchange,
                        string routingKey,
                        [AmqpContentHeaderMapping]
                        IBasicProperties basicProperties,
                        [AmqpContentBodyMapping]
                        byte[] body);
```

In the example above, whenever a `basic.deliver` method arrives, it will be dispatched, as it arrives, to the hand-written implementation of HandleBasicDeliver, and will not be assumed to be the reply to an earlier RPC-style request.

Note that `AmqpContentHeaderMapping` and `AmqpContentBodyMapping` are permitted on the arguments of an asynchronous event handler.

9.4.7. Overriding behaviour for particular specification versions

In some versions of the specification, a particular field, method or content header may not be supported. Methods and parameters can be marked to cause particular implementations to complain if these features are used by applications:

```
[AmqpUnsupported("RabbitMQ.Client.Framing.v0_8qpid")]
void QueueUnbind(string queue,
                 string exchange,
                 string routingKey,
                 IDictionary arguments);
```

In the example above, `queue.unbind` is marked as unsupported by the QPid-specific variant of the 0-8 protocol. Clients using the QPid-specific 0-8 `IProtocol` implementation will, at runtime, receive a thrown `UnsupportedMethodException` if they call QueueUnbind.

```
[AmqpForceOneWay]
[AmqpMethodMapping(null, "basic", "consume")]
void _Private_BasicConsume(ushort ticket,
                           string queue,
                           string consumerTag,
                           bool noLocal,
                           bool noAck,
                           bool exclusive,
                           bool nowait,
                           [AmqpUnsupported("RabbitMQ.Client.Framing.v0_8")]
                           [AmqpFieldMapping("RabbitMQ.Client.Framing.v0_9",
                                             "filter")]
                           IDictionary arguments);
```

In the example above, the "arguments" parameter is unsupported in standard 0-8, and in 0-9 t is named "filter" instead of "arguments". In all other implementations, it is assumed to be supported and named "arguments".

The parameter to the attribute specifies the specification version to which the attribute should apply, or null if it should apply to any and all specification versions.

Notes

1. See `RabbitMQ.Client.Impl.SessionManager.CheckAutoClose()` for the implementation of this feature.

2. The underlying implementation reason is that the `IBasicConsumer` callbacks are invoked in the connection's thread rather than in the application's thread.

3. Within limits - please see the AMQP specification for precise details of when redelivery is permitted and when and how it is mandated.

4. See the AMQP specification for full details of the meaning of "durable" here

5. Submissions welcome!