Messaging with RabbitMQ and .NET review part 11: various other topics

SEPTEMBER 5, 2016 1 COMMENT (HTTPS://DOTNETCODR.COM/2016/09/05/MESSAGING-WITH-RABBITMQ-AND-NET-REVIEW-PART-11-VARIOUS-OTHER-TOPICS/#COMMENTS)

Introduction

In the previous post (https://dotnetcodr.com/2016/09/01/messaging-with-rabbitmq-and-net-review-part-10-scattergather/) we looked at the scatter/gather message exchange pattern. It is similar to RPC (https://dotnetcodr.com/2016/08/18/messaging-with-rabbitmq-and-net-review-part-7-two-way-messaging/) in that the sender will be expecting a response from the receiver. The main difference is that in this scenario the sender can collect a range of responses from various receivers. The sender will set up a temporary response queue where the receivers can send their responses. This MEP is suitable for scenarios that require 2 way communication with more than a single consumer. An example would be a system where we're sending out a notice to some selected construction companies asking for a price offer. The companies then can respond using the message broker and the temporary response queue.

In this post, which will also finish the series, we'll look at various smaller topics around the RabbitMq client.

Mandatory queuing

If a message cannot be forwarded to any queue from an exchange it's lost by default. In other words the publisher doesn't know whether a message has been relayed to at least one queue. There is a way around that via an overload of the BasicPublish method. We also need to set up an event handler for the BasicReturn event of the IModel object. The event handler is triggered in case there was no matching queue for the message. The following example deliberately sets up an exchange with no queue:

```
1  private static void SetUpDirectExchange()
2  {
3      ConnectionFactory connectionFactory = new ConnectionFactory();
4
5      connectionFactory.Port = 5672;
6      connectionFactory.HostName = "localhost";
7      connectionFactory.UserName = "accountant";
8      connectionFactory.Password = "accountant";
9      connectionFactory.VirtualHost = "accounting";
10
11     IConnection connection = connectionFactory.CreateConnection();
12     IModel channel = connection.CreateModel();
13
14     channel.ExchangeDeclare("no.queue.exchange", ExchangeType.Direct, true, false, null);
15     IBasicProperties properties = channel.CreateBasicProperties();
16     channel.BasicReturn += Channel_BasicReturn;
17
18     channel.BasicPublish("no.queue.exchange", "", true, properties, Encoding.UTF8.GetBytes("This is a
19
20     channel.Close();
21     connection.Close();
22 }
23
24 private static void Channel_BasicReturn(object sender, BasicReturnEventArgs e)
25 {
26     Debug.WriteLine(string.Concat("Queue is missing for the message: ", Encoding.UTF8.GetString(e.Bod
27     Debug.WriteLine(string.Concat("Reply code and text: ", e.ReplyCode, " ", e.ReplyText));
28 }
```

The above example produces the following output in the Debug window:

Queue is missing for the message: This is a message from the RabbitMq .NET driver
Reply code and text: 312 NO_ROUTE

Confirmation from the exchange

The publisher can receive a confirmation from RabbitMq whether a message successfully reached the exchange. Note the following three components in the example code:

- ConfirmSelects: it activates feedback mechanism for the publisher
- The BasicAcks event handler which is called in case the message broker has acknowledged the message from the publisher
- The BasicNacks event handler which is triggered in case RabbitMq for some reason could not acknowledge a message. In this case you can re-send a message if it's of critical importance

```
private static void SetUpDirectExchange()
{
    ConnectionFactory connectionFactory = new ConnectionFactory();

    connectionFactory.Port = 5672;
    connectionFactory.HostName = "localhost";
    connectionFactory.UserName = "accountant";
    connectionFactory.Password = "accountant";
    connectionFactory.VirtualHost = "accounting";

    IConnection connection = connectionFactory.CreateConnection();
    IModel channel = connection.CreateModel();

    channel.ExchangeDeclare("my.first.exchange", ExchangeType.Direct, true, false, null);
    channel.QueueDeclare("my.first.queue", true, false, false, null);
    channel.QueueBind("my.first.queue", "my.first.exchange", "");
    channel.ConfirmSelect();
    channel.BasicAcks += Channel_BasicAcks;
    channel.BasicNacks += Channel_BasicNacks;

    IBasicProperties properties = channel.CreateBasicProperties();
    PublicationAddress address = new PublicationAddress(ExchangeType.Direct, "my.first.exchange", "")
    channel.BasicPublish(address, properties, Encoding.UTF8.GetBytes("This is a message from the Rabb

    channel.Close();
    connection.Close();
}

private static void Channel_BasicNacks(object sender, BasicNackEventArgs e)
{
    Console.WriteLine(string.Concat("Message broker could not acknowledge message with tag: ", e.Deli
}

private static void Channel_BasicAcks(object sender, BasicAckEventArgs e)
{
    Console.WriteLine(string.Concat("Message broker has acknowledged message with tag: ", e.DeliveryT
}
```

Unacknowledged messages

A recurring line of code in our demos was that the message registered by the receiver had to be acknowledged:

```
channel.BasicAck(deliveryTag, false);
```

The reason we did that was that the "noAck" flag in the channel.BasicConsume function was set to false. That configuration ensures that a message is not deleted from the queue until it has been acknowledged by the consumer:

```
channel.BasicConsume("my.first.queue", false, basicConsumer);
```

As soon as that publisher has acknowledged the message it is deleted from the queue. However, what if there is no acknowledgement? It can happen in at least two cases:

- If there's an exception during the message processing then the receiver might want to force resending the message. The message will then be requeued and redelivered
- If the consumer crashes after receiving the message but before sending the acknowledgement then either another consumer instance can receive it or the same instance after rebooting

The "redelivered" flag will be true in both cases so you'll have to consider it in your code. If a message is delivered more than once then there might be something wrong with it.

To actively "unacknowledge" a message use the BasicNack function:

```
1 | channel.BasicNack(deliveryTag, false, true);
```

The first two parameters are the same as for BasicAck, the last one means whether the message should be requeued.

If "noAck" in BasicConsume is set to true then the message will be deleted from the queue as soon as it's been delivered.

Object serialisation

So far we've only sent simple string messages back and forth. However, in reality you'll often need to send objects across the wire. We took up data serialisation in the original series and those posts are still applicable. Keep in mind that the receiver examples in that series are based on an outdated object. Make sure you use the receiver techniques we've seen in this blog.

- Data serialisation I (https://dotnetcodr.com/2014/06/05/rabbitmq-in-net-data-serialisation/)
- Data serialisation II (https://dotnetcodr.com/2014/06/09/rabbitmq-in-net-data-serialisation-ii/)

Large messages

Sometimes we need to send large messages through RabbitMq, although the idea with messaging is that message size should be as little as possible. In case you need a solution for large messages you can refer back to the previous RabbitMq series. Like above, make sure you update the receiver:

RabbitMQ in .NET: handling large messages (https://dotnetcodr.com/2014/06/12/rabbitmq-in-net-handling-large-messages/)

In general you should avoid large messages and message dependencies. This latter means that message A is followed by message B which in turn is followed by message C and the three are correlated, then having one exception can cause an unnecessarily complex message handling logic.

Exception handling

Finally we have the topic of exception handling. The following posts from the previous series…

- Basic error handling in the Receiver (https://dotnetcodr.com/2014/06/16/rabbitmq-in-net-c-basic-error-handling-in-receiver/)
- More complex error handling in the Receiver (https://dotnetcodr.com/2014/06/19/rabbitmq-in-net-c-more-complex-error-handling-in-the-receiver/)

…take up the following topics:

- Rejecting a faulty message in the receiver
- Requeing a faulty message for a retry
- Retries

That's the end of this series. The next series will also be related to messaging: we'll take a look at the MassTransit service bus solution (http://masstransit-project.com/).

View the list of posts on Messaging here (https://dotnetcodr.com/messaging/).
FILED UNDER .NET, MESSAGING TAGGED WITH C#, RABBITMQ

About Andras Nemes
I'm a .NET/Java developer living and working in Stockholm, Sweden.

One Response to *Messaging with RabbitMQ and .NET review part 11: various*

Donny V says:
August 16, 2019 at 2:46 pm
If your using channel.ConfirmSelect(); you also need to call channel.WaitForConfirms(); after channel.BasicPublish(). If not then the method will finish executing before channel.BasicAcks & channel.BasicNacks can fire.

Reply

Create a free website or blog at WordPress.com.