

# RabbitMQ in .NET C#: more complex error handling in the Receiver

JUNE 19, 2014    1 COMMENT ([HTTPS://DOTNETCODR.COM/2014/06/19/RABBITMQ-IN-NET-C-MORE-COMPLEX-ERROR-HANDLING-IN-THE-RECEIVER/#COMMENTS](https://dotnetcodr.com/2014/06/19/RABBITMQ-IN-NET-C-MORE-COMPLEX-ERROR-HANDLING-IN-THE-RECEIVER/#COMMENTS)).

## Introduction

In the [previous part](https://dotnetcodr.com/2014/06/16/rabbitmq-in-net-c-basic-error-handling-in-receiver/) (<https://dotnetcodr.com/2014/06/16/rabbitmq-in-net-c-basic-error-handling-in-receiver/>) on RabbitMQ .NET we looked at ways how to reject a message if there was an exception while handling the message on the Receiver's side. The message could then be discarded or re-queued for a retry. However, the exception handling logic was very primitive in that the same message could potentially be thrown at the receiver infinitely causing a traffic jam in the messages.

This post builds upon the basics of RabbitMQ in .NET. If you are new to this topic you should check out all the previous posts listed on [this](https://dotnetcodr.com/messaging/) (<https://dotnetcodr.com/messaging/>) page. I won't provide any details on bits of code that we've gone through before.

Most of the posts on RabbitMQ on this blog are based on the work of RabbitMQ guru [Michael Stephenson](http://geekswithblogs.net/michaelstephenson/Default.aspx) (<http://geekswithblogs.net/michaelstephenson/Default.aspx>).

So we cannot just keep retrying forever. We can instead finally discard the message after a certain amount of retries or depending on what kind of exception was encountered.

The logic around retries must be implemented in the receiver as there's no simple method in RabbitMQ .NET, like "BasicRetry". Why should there be anyway? Retry strategies can be very diverse so it's easier to let the receiver handle it.

The strategy here is to reject the message without re-queuing it. We'll then create a new message based on the one that caused the exception and attach an integer value to it indicating the number of retries. Then depending on a maximum ceiling we either create yet another message for re-queuing or discard it altogether.

We'll build on the demo we started on in the previous post referred to above so have it ready.

## Demo

We'll reuse the queue from the previous post which we called "BadMessageQueue". We'll also reuse the code in BadMessageSender as there's no variation on the Sender side.

BadMessageReceiver will however handle the messages in a different way. Currently there's a method called ReceiveBadMessages which is called upon from Main. Comment out that method call. Insert the following method in ReceiveBadMessages.Program.cs and call it from Main:

```

1 private static void ReceiveBadMessageExtended(IModel model)
2 {
3     model.BasicQos(0, 1, false);
4     QueueingBasicConsumer consumer = new QueueingBasicConsumer(model);
5     model.BasicConsume(RabbitMqService.BadMessageBufferedQueue, false, consumer);
6     string customRetryHeaderName = "number-of-retries";
7     int maxNumberOfRetries = 3;
8     while (true)
9     {
10         BasicDeliverEventArgs deliveryArguments = consumer.Queue.Dequeue() as BasicDeliverEventArgs;
11         String message = Encoding.UTF8.GetString(deliveryArguments.Body);
12         Console.WriteLine("Message from queue: {0}", message);
13         Random random = new Random();
14         int i = random.Next(0, 3);
15         int retryCount = GetRetryCount(deliveryArguments.BasicProperties, customRetryHeaderName);
16         if (i == 2) //no exception, accept message
17         {
18             Console.WriteLine("Message {0} accepted. Number of retries: {1}", message, retryCount);
19             model.BasicAck(deliveryArguments.DeliveryTag, false);
20         }
21         else //simulate exception: accept message, but create copy and throw back
22         {
23             if (retryCount < maxNumberOfRetries)
24             {
25                 Console.WriteLine("Message {0} has thrown an exception. Current number of retries: {1}
26                 IBasicProperties propertiesForCopy = model.CreateBasicProperties();
27                 IDictionary<string, object> headersCopy = CopyHeaders(deliveryArguments.BasicProperties
28                 propertiesForCopy.Headers = headersCopy;
29                 propertiesForCopy.Headers[customRetryHeaderName] = ++retryCount;
30                 model.BasicPublish(deliveryArguments.Exchange, deliveryArguments.RoutingKey, properti
31                 model.BasicAck(deliveryArguments.DeliveryTag, false);
32                 Console.WriteLine("Message {0} thrown back at queue for retry. New retry count: {1}",
33             }
34             else //must be rejected, cannot process
35             {
36                 Console.WriteLine("Message {0} has reached the max number of retries. It will be reje
37                 model.BasicReject(deliveryArguments.DeliveryTag, false);
38             }
39         }
40     }
41 }

```

...where CopyHeaders and GetRetryCount look as follows:

```

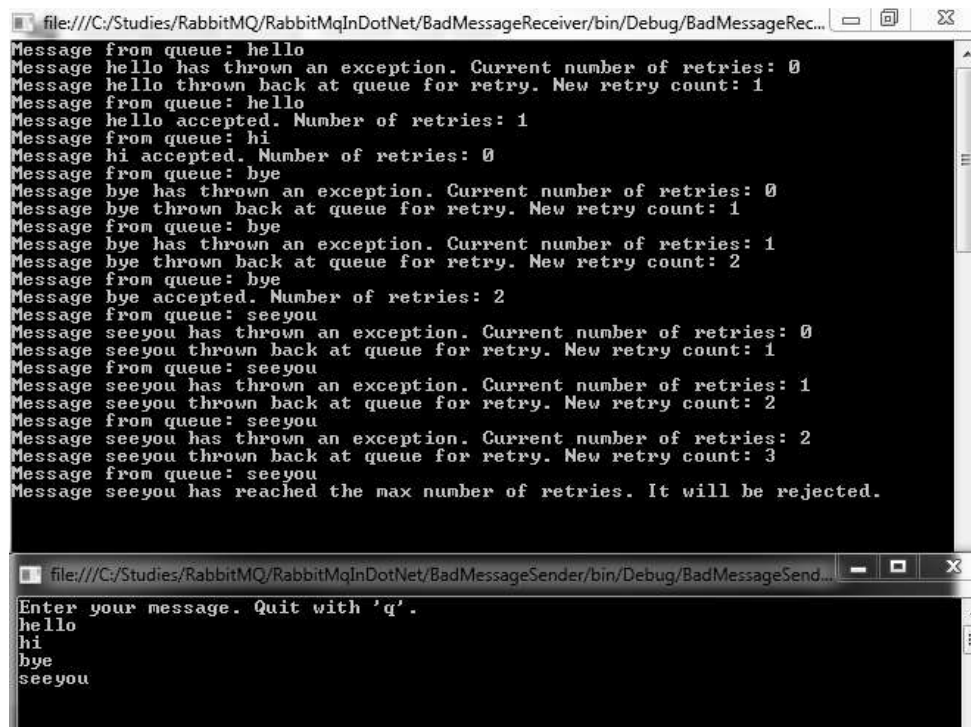
1 private static IDictionary<string, object> CopyHeaders(IBasicProperties originalProperties)
2 {
3     IDictionary<string, object> dict = new Dictionary<string, object>();
4     IDictionary<string, object> headers = originalProperties.Headers;
5     if (headers != null)
6     {
7         foreach (KeyValuePair<string, object> kvp in headers)
8         {
9             dict[kvp.Key] = kvp.Value;
10        }
11    }
12
13    return dict;
14 }
15
16 private static int GetRetryCount(IBasicProperties messageProperties, string countHeader)
17 {
18     IDictionary<string, object> headers = messageProperties.Headers;
19     int count = 0;
20     if (headers != null)
21     {
22         if (headers.ContainsKey(countHeader))
23         {
24             string countAsString = Convert.ToString( headers[countHeader]);
25             count = Convert.ToInt32(countAsString);
26         }
27     }
28
29     return count;
30 }

```

Let's see what's going on here. We define a custom header to store the number of retries for a message. We also set an upper limit of 3 on the number of retries. Then we accept the messages in the usual way. A random number between 0 and 3 is generated – where the upper limit is exclusive – to decide whether to simulate an exception or not. If this number is 2 then we accept and acknowledge the message, so there's a higher probability of "throwing an exception" just to make this demo more interesting. We also extract the current number of retries using the GetRetryCount method. This helper method simply checks the headers of the message for the presence of the custom retry count header.

If we simulate an exception then we need to check if the current retry count has reached the max number of retries. If not then the exciting new stuff begins. We create a new message where we copy the elements of the original message. We also set the new value of the retry count header. We send the message copy back to where it came from and acknowledge the original message. Otherwise if the max number of retries has been reached we reject the message completely using the BasicReject method we saw in the previous part.

Run both the Sender and Receiver apps and start sending messages from the Sender. Depending on the random number generated in the Receiver you'll see a differing number of retries but you may get something like this:



```
file:///C:/Studies/RabbitMQ/RabbitMqInDotNet/BadMessageReceiver/bin/Debug/BadMessageRec...
Message from queue: hello
Message hello has thrown an exception. Current number of retries: 0
Message hello thrown back at queue for retry. New retry count: 1
Message from queue: hello
Message hello accepted. Number of retries: 1
Message from queue: hi
Message hi accepted. Number of retries: 0
Message from queue: bye
Message bye has thrown an exception. Current number of retries: 0
Message bye thrown back at queue for retry. New retry count: 1
Message from queue: bye
Message bye has thrown an exception. Current number of retries: 1
Message bye thrown back at queue for retry. New retry count: 2
Message from queue: bye
Message bye accepted. Number of retries: 2
Message from queue: seeyou
Message seeyou has thrown an exception. Current number of retries: 0
Message seeyou thrown back at queue for retry. New retry count: 1
Message from queue: seeyou
Message seeyou has thrown an exception. Current number of retries: 1
Message seeyou thrown back at queue for retry. New retry count: 2
Message from queue: seeyou
Message seeyou has thrown an exception. Current number of retries: 2
Message seeyou thrown back at queue for retry. New retry count: 3
Message from queue: seeyou
Message seeyou has reached the max number of retries. It will be rejected.

file:///C:/Studies/RabbitMQ/RabbitMqInDotNet/BadMessageSender/bin/Debug/BadMessageSend...
Enter your message. Quit with 'q'.
hello
hi
bye
seeyou
```

(<https://dotnetcodr.files.wordpress.com/2014/05/advanced-retry-console-output.png>).

We can see the following here:

- Message hello was rejected at first and then accepted after 1 retry
- Message hi was accepted immediately
- Message bye was accepted after 2 retries
- Message seeyou was rejected completely

So we've seen how to add some more logic into how to handle exceptions.

Other considerations and extensions:

- You can specify different max retries depending on the exception type. In that case you can add the exception type to the headers as well
- You might consider storing the retry count somewhere else than the message itself, e.g. within the Receiver – the advantage of storing the retry count in the message is that if you have multiple receivers waiting for messages from the same queue then they will all have access to the retry property
- If there's a dependency between messages then exception handling becomes a bigger challenge: if message B depends on message A and message A throws an exception, what do we do with message B? You can force related messages to be processed in an ordered fashion which will have a negative impact on the message throughput. On the other hand you may simply ignore this scenario if it's not important enough for your case – “enough” depends on the cost of slower message throughput versus the cost of an exception in interdependent messages. Somewhere between these two extremes you can decide to keep the order of related messages only and let all others be delivered normally. In this case you can put the sequence number, such as “5/10” in the header so that the receiver can check if all messages have come in correctly. If you have multiple receivers then the sequence number must be stored externally so that all receivers will have access to the same information. Otherwise you can have a separate queue or even a separate RabbitMQ instance for related messages in case the proportion of related messages in total number of messages is small.

View the list of posts on Messaging [here](https://dotnetcodr.com/messaging/) (<https://dotnetcodr.com/messaging/>).

FILED UNDER [.NET, MESSAGING](#) TAGGED WITH [C#, EXCEPTION HANDLING, RABBITMQ](#)

**About Andras Nemes**

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

## One Response to *RabbitMQ in .NET C#: more complex error handling in the Receiver*

**ANU PRAKASH** says:



September 29, 2019 at 11:50 am

It will be helpful if you can share the full source code link for these post.

**Reply**

Create a free website or blog at WordPress.com.

Advertisements



In-store shopping

In-store pickup

REPORT THIS AD