RabbitMQ tutorial - Reliable Publishing with Publisher Confirms

Publisher Confirms

INFO

Prerequisites

This tutorial assumes RabbitMQ is installed and running on `localhost` on the standard port (5672). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

If you're having trouble going through this tutorial you can contact us through GitHub Discussions or RabbitMQ community Discord.

Publisher confirms are a RabbitMQ extension to implement reliable publishing. When publisher confirms are enabled on a channel, messages the client publishes are confirmed asynchronously by the broker, meaning they have been taken care of on the server side.

(using the .NET client)

Overview

In this tutorial we're going to use publisher confirms to make sure published messages have safely reached the broker. We will cover several strategies to using publisher confirms and explain their pros and cons.

Enabling Publisher Confirms on a Channel

Publisher confirms are a RabbitMQ extension to the AMQP 0.9.1 protocol, so they are not enabled by default. Publisher confirms are enabled at the channel level with the `ConfirmSelect` method:

```
var channel = connection.CreateModel();
channel.ConfirmSelect();
```

This method must be called on every channel that you expect to use publisher confirms. Confirms should be enabled just once, not for every message published.

Strategy #1: Publishing Messages Individually

Let's start with the simplest approach to publishing with confirms, that is, publishing a message and waiting synchronously for its confirmation:

```
while (ThereAreMessagesToPublish())
{
    byte[] body = ...;
    IBasicProperties properties = ...;
    channel.BasicPublish(exchange, queue, properties, body);
    // uses a 5 second timeout
    channel.WaitForConfirmsOrDie(TimeSpan.FromSeconds(5));
}
```

In the previous example we publish a message as usual and wait for its confirmation with the `Channel#WaitForConfirmsOrDie(TimeSpan)` method. The method returns as soon as the message has been confirmed. If the message is not confirmed within the timeout or if it is nack-ed (meaning the broker could not take care of it for some reason), the method will throw an exception. The handling of the exception usually consists in logging an error message and/or retrying to send the message.

Different client libraries have different ways to synchronously deal with publisher confirms, so make sure to read carefully the documentation of the client you are using.

This technique is very straightforward but also has a major drawback: it **significantly slows down publishing**, as the confirmation of a message blocks the publishing of all subsequent messages. This approach is not going to deliver throughput of more than a few hundreds of published messages per second. Nevertheless, this can be good enough for some applications.

> **Are Publisher Confirms Asynchronous?**
>
> We mentioned at the beginning that the broker confirms published messages asynchronously but in the first example the code waits synchronously until the message is confirmed. The client actually receives confirms asynchronously and unblocks the call to `WaitForConfirmsOrDie` accordingly. Think of `WaitForConfirmsOrDie` as a synchronous helper which relies on asynchronous notifications under the hood.

Strategy #2: Publishing Messages in Batches

To improve upon our previous example, we can publish a batch of messages and wait for this whole batch to be confirmed. The following example uses a batch of 100:

```
var batchSize = 100;
var outstandingMessageCount = 0;
while (ThereAreMessagesToPublish())
{
    byte[] body = ...;
    IBasicProperties properties = ...;
    channel.BasicPublish(exchange, queue, properties, body);
    outstandingMessageCount++;
    if (outstandingMessageCount == batchSize)
    {
        channel.WaitForConfirmsOrDie(TimeSpan.FromSeconds(5));
        outstandingMessageCount = 0;
    }
}
if (outstandingMessageCount > 0)
{
    channel.WaitForConfirmsOrDie(TimeSpan.FromSeconds(5));
}
```

Waiting for a batch of messages to be confirmed improves throughput drastically over waiting for a confirm for individual message (up to 20-30 times with a remote RabbitMQ node). One drawback is that we do not know exactly what went wrong in case of failure, so we may have to keep a whole batch in memory to log something meaningful or to re-publish the messages. And this solution is still synchronous, so it blocks the publishing of messages.

Strategy #3: Handling Publisher Confirms Asynchronously

The broker confirms published messages asynchronously, one just needs to register a callback on the client to be notified of these confirms:

```
var channel = connection.CreateModel();
channel.ConfirmSelect();
channel.BasicAcks += (sender, ea) =>
{
  // code when message is confirmed
};
channel.BasicNacks += (sender, ea) =>
{
  //code when message is nack-ed
};
```

There are 2 callbacks: one for confirmed messages and one for nack-ed messages (messages that can be considered lost by the broker). Both callbacks have a corresponding `EventArgs` parameter (`ea`) containing a:

- delivery tag: the sequence number identifying the confirmed or nack-ed message. We will see shortly how to correlate it with the published message.
- multiple: this is a boolean value. If false, only one message is confirmed/nack-ed, if true, all messages with a lower or equal sequence number are confirmed/nack-ed.

The sequence number can be obtained with `Channel#NextPublishSeqNo` before publishing:

```
var sequenceNumber = channel.NextPublishSeqNo;
channel.BasicPublish(exchange, queue, properties, body);
```

A simple way to correlate messages with sequence number consists in using a dictionary. Let's assume we want to publish strings because they are easy to turn into an array of bytes for publishing. Here is a code sample that uses a dictionary to correlate the publishing sequence number with the string body of the message:

```
var outstandingConfirms = new ConcurrentDictionary<ulong, string>();
// ... code for confirm callbacks will come later
var body = "...";
outstandingConfirms.TryAdd(channel.NextPublishSeqNo, body);
channel.BasicPublish(exchange, queue, properties, Encoding.UTF8.GetBytes(body));
```

The publishing code now tracks outbound messages with a dictionary. We need to clean this dictionary when confirms arrive and do something like logging a warning when messages are nack-ed:

```
var outstandingConfirms = new ConcurrentDictionary<ulong, string>();

void CleanOutstandingConfirms(ulong sequenceNumber, bool multiple)
{
    if (multiple)
    {
        var confirmed = outstandingConfirms.Where(k => k.Key <= sequenceNumber);
        foreach (var entry in confirmed)
        {
            outstandingConfirms.TryRemove(entry.Key, out _);
        }
    }
    else
    {
        outstandingConfirms.TryRemove(sequenceNumber, out _);
    }
}

channel.BasicAcks += (sender, ea) => CleanOutstandingConfirms(ea.DeliveryTag,
ea.Multiple);
channel.BasicNacks += (sender, ea) =>
{
    outstandingConfirms.TryGetValue(ea.DeliveryTag, out string body);
    Console.WriteLine($"Message with body {body} has been nack-ed. Sequence
number: {ea.DeliveryTag}, multiple: {ea.Multiple}");
    CleanOutstandingConfirms(ea.DeliveryTag, ea.Multiple);
};

// ... publishing code
```

The previous sample contains a callback that cleans the dictionary when confirms arrive. Note this callback handles both single and multiple confirms. This callback is used when confirms arrive (`Channel#BasicAcks`). The callback for nack-ed messages retrieves the message body and issues a warning. It then re-uses the previous callback to clean the dictionary of outstanding confirms (whether messages are confirmed or nack-ed, their corresponding entries in the dictionary must be removed.)

> **How to Track Outstanding Confirms?**
>
> Our samples use a `ConcurrentDictionary` to track outstanding confirms. This data structure is convenient for several reasons. It allows to easily correlate a sequence number with a message (whatever the message data is) and to easily clean the entries up to a given sequence id (to handle multiple confirms/nacks). At last, it supports concurrent access, because confirm callbacks are called in a thread owned by the client library, which should be kept different from the publishing thread.
>
> There are other ways to track outstanding confirms than with a sophisticated dictionary implementation, like using a simple concurrent hash table and a variable to track the lower bound of the publishing sequence, but they are usually more involved and do not belong to a tutorial.

To sum up, handling publisher confirms asynchronously usually requires the following steps:

- provide a way to correlate the publishing sequence number with a message.
- register confirm listeners on the channel to be notified when publisher acks/nacks arrive to perform the appropriate actions, like logging or re-publishing a nack-ed message. The sequence-number-to-message correlation mechanism may also require some cleaning during this step.
- track the publishing sequence number before publishing a message.

> **Re-publishing nack-ed Messages?**
>
> It can be tempting to re-publish a nack-ed message from the corresponding callback but this should be avoided, as confirm callbacks are dispatched in an I/O thread where channels are not supposed to do operations. A better solution consists in enqueuing the message in an in-memory queue which is polled by a publishing thread. A class like `ConcurrentQueue` would be a good candidate to transmit messages between the confirm callbacks and a publishing thread.

Summary

Making sure published messages made it to the broker can be essential in some applications. Publisher confirms are a RabbitMQ feature that helps to meet this requirement. Publisher confirms are asynchronous in nature but it is also possible to handle them synchronously. There is no definitive way to implement publisher confirms, this usually comes down to the constraints in the application and in the overall system. Typical techniques are:

- publishing messages individually, waiting for the confirmation synchronously: simple, but very limited throughput.
- publishing messages in batch, waiting for the confirmation synchronously for a batch: simple, reasonable throughput, but hard to reason about when something goes wrong.
- asynchronous handling: best performance and use of resources, good control in case of error, but can be involved to implement correctly.

Putting It All Together

The `PublisherConfirms.cs` class contains code for the techniques we covered. We can compile it, execute it as-is and see how they each perform:

```
dotnet run
```

The output will look like the following:

```
Published 50,000 messages individually in 5,549 ms
Published 50,000 messages in batch in 2,331 ms
Published 50,000 messages and handled confirms asynchronously in 4,054 ms
```

The output on your computer should look similar if the client and the server sit on the same machine. Publishing messages individually performs poorly as expected, but the results for asynchronously handling are a bit disappointing compared to batch publishing.

Publisher confirms are very network-dependent, so we're better off trying with a remote node, which is more realistic as clients and servers are usually not on the same machine in production. `PublisherConfirms.cs` can easily be changed to use a non-local node:

```
private static IConnection CreateConnection()
{
    var factory = new ConnectionFactory { HostName = "remote-host", UserName =
"remote-host", Password = "remote-password" };
    return factory.CreateConnection();
}
```

Recompile the class, execute it again, and wait for the results:

```
Published 50,000 messages individually in 231,541 ms
Published 50,000 messages in batch in 7,232 ms
Published 50,000 messages and handled confirms asynchronously in 6,332 ms
```

We see publishing individually now performs terribly. But with the network between the client and the server, batch publishing and asynchronous handling now perform similarly, with a small advantage for asynchronous handling of the publisher confirms.

Remember that batch publishing is simple to implement, but does not make it easy to know which message(s) could not make it to the broker in case of negative publisher acknowledgment. Handling publisher confirms asynchronously is more involved to implement but provide better granularity and better control over actions to perform when published messages are nack-ed.