Messaging with RabbitMQ and .NET review part 10: scatter/gather

SEPTEMBER 1, 2016 1 COMMENT (HTTPS://DOTNETCODR.COM/2016/09/01/MESSAGING-WITH-RABBITMQ-AND-NET-REVIEW-PART-10-SCATTERGATHER/#COMMENTS)

Introduction

In the previous post (https://dotnetcodr.com/2016/08/29/messaging-with-rabbitmq-and-net-review-part-9-headers/) we looked at message filtering using message headers in RabbitMq. The Headers exchange pattern is very similar to Topics we saw in this post (https://dotnetcodr.com/2016/08/25/messaging-with-rabbitmq-and-net-review-part-8-routing-and-topics/). The sender sends a message of type Headers to RabbitMq. The message is routed based on the header values. All queues with a matching key will receive the message. We can specify more than one header and a rule that says if all headers or just one of them must match. The headers come in key-value pairs such as "category:vehicle", "type:car". Headers allow us to fine-tune our routing rules to a great extent.

In this post we'll look at one more message exchange pattern called scatter/gather. It is similar to two-way messaging (https://dotnetcodr.com/2016/08/18/messaging-with-rabbitmq-and-net-review-part-7-two-way-messaging/) but the publisher will get the responses from multiple consumers.

Scatter/gather

This pattern is similar to the RPC message exchange pattern 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. It's possible to implement this pattern using any exchange type: fanout, direct, headers and topic depending on how you've set up the exchange/queue binding. In other words there is no specific exchange type in RabbitMq that specifically corresponds to scatter/gather, we have to do some extra coding ourselves. You can also specify a routing key in the binding like we saw before.

I think this is definitely a pattern which can be widely used in real applications out there that require 2 way communication with more than a single consumer. Consider that you send out a request to construction companies asking for a price offer. The companies can then respond using the message broker through the temporary response queue.

We'll re-use several ideas and bits of code from the RPC pattern so make sure you understand the basics of that MEP as well. We'll be working in the same demo .NET console application as before.

Here's the code that sets up the publisher:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using RabbitMQ.Client;
using RabbitMQ.Client.MessagePatterns;
using RabbitMQ.Client.Events;
using System.Threading;

namespace RabbitMqNetTests
{
    class Program
    {
        static void Main(string[] args)
        {
            RunScatterGatherQueue();
            Console.ReadKey();
        }

        private static void RunScatterGatherQueue()
        {
            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.QueueDeclare("mycompany.queues.scattergather.a", true, false, false, null);
            channel.QueueDeclare("mycompany.queues.scattergather.b", true, false, false, null);
            channel.QueueDeclare("mycompany.queues.scattergather.c", true, false, false, null);
            channel.ExchangeDeclare("mycompany.exchanges.scattergather", ExchangeType.Fanout, true, f
            channel.QueueBind("mycompany.queues.scattergather.a", "mycompany.exchanges.scattergather"
            channel.QueueBind("mycompany.queues.scattergather.b", "mycompany.exchanges.scattergather"
            channel.QueueBind("mycompany.queues.scattergather.c", "mycompany.exchanges.scattergather"
            SendScatterGatherMessages(connection, channel, 3);
        }

        private static void SendScatterGatherMessages(IConnection connection, IModel channel, int min
        {
            List<string> responses = new List<string>();
            string rpcResponseQueue = channel.QueueDeclare().QueueName;
            string correlationId = Guid.NewGuid().ToString();

            IBasicProperties basicProperties = channel.CreateBasicProperties();
            basicProperties.ReplyTo = rpcResponseQueue;
            basicProperties.CorrelationId = correlationId;
            Console.WriteLine("Enter your message and press Enter.");

            string message = Console.ReadLine();
            byte[] messageBytes = Encoding.UTF8.GetBytes(message);
            channel.BasicPublish("mycompany.exchanges.scattergather", "", basicProperties, messageByt

            EventingBasicConsumer scatterGatherEventingBasicConsumer = new EventingBasicConsumer(chan
            scatterGatherEventingBasicConsumer.Received += (sender, basicDeliveryEventArgs) =>
            {
                IBasicProperties props = basicDeliveryEventArgs.BasicProperties;
                channel.BasicAck(basicDeliveryEventArgs.DeliveryTag, false);
                if (props != null
                    && props.CorrelationId == correlationId)
                {
                    string response = Encoding.UTF8.GetString(basicDeliveryEventArgs.Body);
                    Console.WriteLine("Response: {0}", response);
                    responses.Add(response);
                    if (responses.Count >= minResponses)
                    {
```

```
                        Console.WriteLine(string.Concat("Responses received from consumers: ", string
                        channel.Close();
                        connection.Close();
                    }
                }
            };
            channel.BasicConsume(rpcResponseQueue, false, scatterGatherEventingBasicConsumer);
        }
    }
}
```


We're simulating a scenario with 3 expected consumers. Each consumer receives its own queue where they can listen – we'll soon see the receiver's code as well:

- mycompany.queues.scattergather.a
- mycompany.queues.scattergather.b
- mycompany.queues.scattergather.c

We also declare a fanout exchange called "mycompany.exchanges.scattergather" and then bind each queue to it.

Much of the SendScatterGatherMessages function looks familiar from the discussion on the RPC MEP. However, there are some key differences. The function allows us to specify a minimum amount of responses we're willing to wait for. In our example this number is 3. We want to collect the responses in the "responses" string list. We build a temporary response queue using the QueueDeclare().QueueName property. Then we publish our first message to the mycompany.exchanges.scattergather exchange and wait for the responses in the implemented Received event handler of scatterGatherEventingBasicConsumer. We simply print the response and add it to the list of responses. If the minimum number of responses has been reached then we close the channel and the connection.

We need to build 3 consumers in order to demo this MEP, i.e. one consumer for each queue. You can add 3 console projects to the demo solution. I've named mine as follows:

- RabbitMq.ScatterGather.Receiver.A
- RabbitMq.ScatterGather.Receiver.B
- RabbitMq.ScatterGather.Receiver.C

All of these must have a reference to the RabbitMq client library from NuGet. Here's the program code for consumer A:

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace RabbitMq.ScatterGather.Receiver
{
    class Program
    {
        static void Main(string[] args)
        {
            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.BasicQos(0, 1, false);
            EventingBasicConsumer eventingBasicConsumer = new EventingBasicConsumer(channel);
            string consumerId = "A";
            Console.WriteLine(string.Concat("Consumer ", consumerId, " up and running, waiting for th
            eventingBasicConsumer.Received += (sender, basicDeliveryEventArgs) =>
            {
                string message = Encoding.UTF8.GetString(basicDeliveryEventArgs.Body);
                channel.BasicAck(basicDeliveryEventArgs.DeliveryTag, false);
                Console.WriteLine("Message: {0} {1}", message, " Enter your response: ");
                string response = string.Concat("Consumer ID: ", consumerId, ", bid: ", Console.ReadL
                IBasicProperties replyBasicProperties = channel.CreateBasicProperties();
                replyBasicProperties.CorrelationId = basicDeliveryEventArgs.BasicProperties.Correlati
                byte[] responseBytes = Encoding.UTF8.GetBytes(response);
                channel.BasicPublish("", basicDeliveryEventArgs.BasicProperties.ReplyTo, replyBasicPr
                channel.Close();
                connection.Close();
            };
            channel.BasicConsume("mycompany.queues.scattergather.a", false, eventingBasicConsumer);
        }
    }
}
```


The only new thing here is that we close the channel and the connection as soon as the receiver has responded. This will also close the execution of the console application, i.e. the console windows will close down immediately. In other words we only let the consumer place a single bid.

The implementation of receivers B and C are identical with 2 differences:

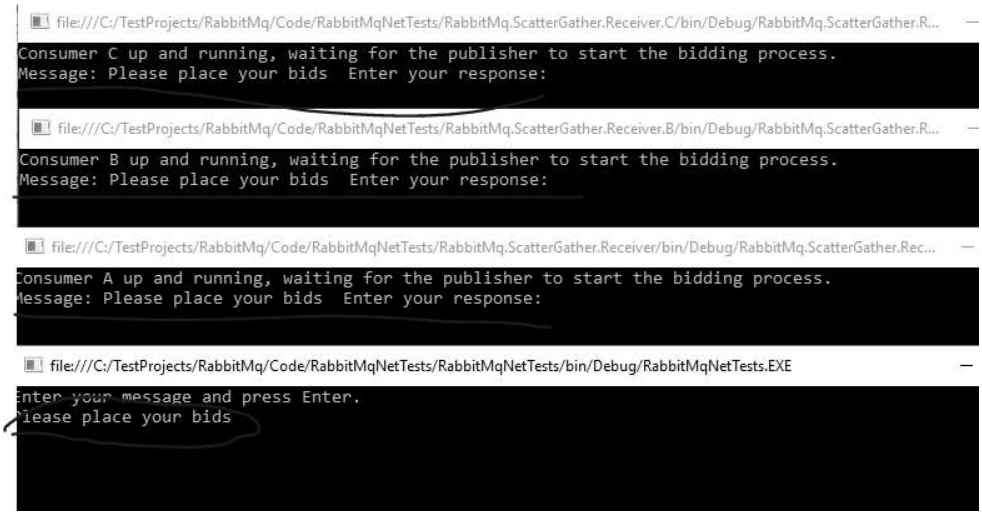
- The variable consumerId will be “B” and “C” respectively
- The queue to be consumed in the BasicConsume method will be mycompany.queues.scattergather.b and mycompany.queues.scattergather.c

Otherwise the code is the same as above.

Start the project which includes the publisher’s implementation. Code execution will stop at the “Enter your message and press Enter.” prompt. Don’t write anything there yet. Then start each consumer using the right-click, Debug, Start new instance technique we saw before. At this point you should have 4 command prompts up and running:

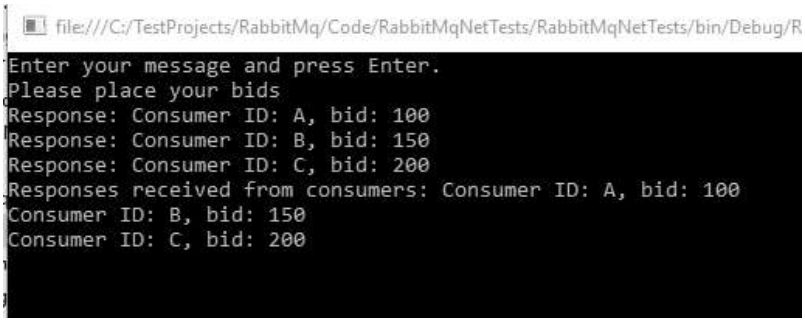
(https://dotnetcodr.files.wordpress.com/2016/08/one-publisher-and-three-consumers-shown-for-the-scatter-gather-demo-in-rabbitmq.png)

Write some message like "Please enter your bids" in the publisher's window and press Enter. The message should appear for each consumer as well:



(https://dotnetcodr.files.wordpress.com/2016/08/scatter-gather-publishers-message-relayed-to-each-consumer-in-rabbitmq.png)

Then start responding to the publisher from each consumer one by one, e.g. consumer A replies with "100", B with "150" and C with "200". The publisher will show each response and then all 3 responses as soon as they are in. You'll also see that the consumers have closed down. Here's the publisher's output after the process:



(https://dotnetcodr.files.wordpress.com/2016/08/scatter-gather-publisher-shows-all-consumer-responses-in-rabbitmq.png)

At this point the publisher's window hasn't closed down due to the Console.ReadKey() in the Main method. Otherwise the channel and connection to RabbitMq have been terminated.

This was a basic implementation of the scatter/gather message exchange pattern in RabbitMq. The next post (https://dotnetcodr.com/2016/09/05/messaging-with-rabbitmq-and-net-review-part-11-various-other-topics/) will wrap up the RabbitMq basics series with some remaining topics.

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 10: scatter/gather*

Annonymous says:

March 23, 2019 at 11:27 am

In your case you are simply posting the responses to screen. How would you return the responses to the calling application? I can do it using QueueingBasicConsumer, however QueueingBasicConsumer is now depreciated. Thanks.

Reply