Exercises in .NET with Andras Nemes

Tips and tricks in C# .NET

Messaging with RabbitMQ and .NET review part 7: two way messaging

AUGUST 18, 2016 LEAVE A COMMENT (HTTPS://DOTNETCODR.COM/2016/08/18/MESSAGING-WITH-RABBITMQ-AND-NET-REVIEW-PART-7-TWO-WAY-MESSAGING/#RESPOND)

Introduction

In the previous post (https://dotnetcodr.com/2016/08/15/messaging-with-rabbitmq-and-net-review-part-6-the-fanout-exchange-type/) we looked at the fanout exchange type. This exchange type corresponds to the publish-subscribe message exchange pattern. This MEP is very similar to one-way messaging. The major difference is that there can be multiple queues bound to an exchange. The incoming messages are forwarded to every queue where each queue is monitored by a listener. This exchange type is suitable for cases where you know that there will be multiple listeners.

So far we've looked at patterns where the publisher sent a message to a queue and didn't care about any response back from the consumer. In this post we'll see how to set up two-way messaging where a publisher receives a response from the consumer.

Remote procedure calls (RPC)

RPC is slightly different from the previous MEPs in that there's a response queue involved. The sender sends an initial message to a destination queue via the default exchange. The message properties include a temporary queue where the consumer can reply. The response queue will be dynamically created by the sender. The receiver processes the message and responds using the response queue extracted from the message properties. The sender then processes the response. In this scenario the publisher will also need a consumer class so that it can process the responses. Both parties will also need to acknowledge the messages they receive. Hence the publisher will acknowledge the response from the sender. It's important to note that we'll have two queues: a "normal" durable queue where the publisher can send messages to the receiver and then a temporary one where the receiver can send the response. The receiver will be listening on the fixed queue and the publisher on the temporary one.

Note that this setup is not mandatory for two way messaging. You can use a dedicated exchange to route the messages. Moreover, the response queue can be a fixed queue. Hence the usage of the default nameless exchange, called the "(AMQP default)" in the management GUI and the temporary response queue are not obligatory. However, it is probably not necessary to have a dedicated exchange for this purpose and it's good to know how to use default exchange as well. Furthermore, the usage of temporary queues which disappear after all channels using it are closed is also important knowledge.

We've been working with a demo console application in Visual Studio in this series and will continue to do so. We currently have a console project in it that includes all code for the sender. We'll build upon that to set up the publisher in the RPC scenario.

Here's the publisher's part of the code:

```csharp
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)
        {
            RunRpcQueue();
        }

        private static void RunRpcQueue()
        {
            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.rpc", true, false, false, null);
            SendRpcMessagesBackAndForth(channel);

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

        private static void SendRpcMessagesBackAndForth(IModel channel)
        {
            string rpcResponseQueue = channel.QueueDeclare().QueueName;

            string correlationId = Guid.NewGuid().ToString();
            string responseFromConsumer = null;

            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.queues.rpc", basicProperties, messageBytes);

            EventingBasicConsumer rpcEventingBasicConsumer = new EventingBasicConsumer(channel);
            rpcEventingBasicConsumer.Received += (sender, basicDeliveryEventArgs) =>
            {
                IBasicProperties props = basicDeliveryEventArgs.BasicProperties;
                if (props != null
                    && props.CorrelationId == correlationId)
                {
                    string response = Encoding.UTF8.GetString(basicDeliveryEventArgs.Body);
                    responseFromConsumer = response;
                }
                channel.BasicAck(basicDeliveryEventArgs.DeliveryTag, false);
                Console.WriteLine("Response: {0}", responseFromConsumer);
                Console.WriteLine("Enter your message and press Enter.");
                message = Console.ReadLine();
                messageBytes = Encoding.UTF8.GetBytes(message);
                channel.BasicPublish("", "mycompany.queues.rpc", basicProperties, messageBytes);
            };
```

```
                channel.BasicConsume(rpcResponseQueue, false, rpcEventingBasicConsumer);
            }

            private static void SetUpFanoutExchange()
            {
                //code ignored from previous posts
            }

            private static void SetUpDirectExchange()
            {
                //code ignored from previous posts
            }
        }
    }
```

The RunRpcQueue is standard code by now. We declare a queue called "mycompany.queues.rpc". This queue will be used as the default queue by the sender to send messages. The response queue will be dynamically set up.

The first section of SendRpcMessagesBackAndForth is also easy to follow. We assign a correlation ID and a reply-to address to the message. The response queue will be set up dynamically. The QueueDeclare() method without any parameter will create a temporary non-durable response queue. The name of the temporary queue will be randomly generated, e.g. "amq.gen-3tj4jtzMauwolYqc7CUj9g". While you're running the demo in a bit you can check the list of queues in the RabbitMQ management UI. The temporary queue will be available as long as the sender is running. After that it will be removed automatically.

We create a message correlation ID to be able to match the sender's message to the response from the receiver. If the receiver is responding to another message then it will be ignored. This is again not mandatory, but a correlation ID ensures that we're communicating with the right consumer. We then set up the IBasicProperties object and specify the temporary queue name to reply to and the correlation ID. Next we publish the message using BasicPublish like before. Note the first empty string parameter in BasicPublish. It denotes the nameless default AMQP exchange.

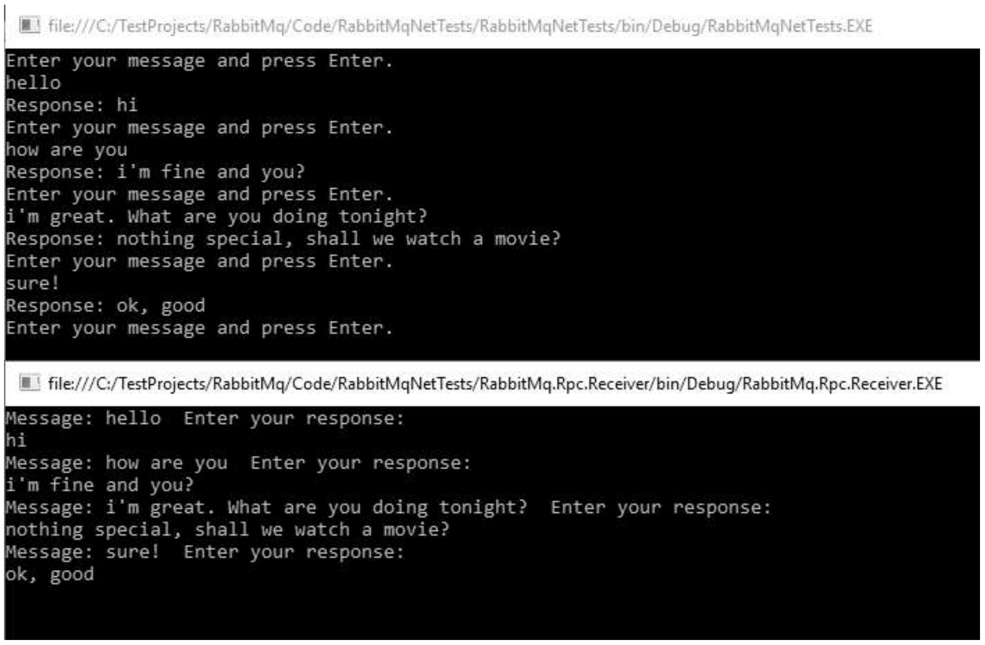
Then we enter something that only receivers have done up to now: listen. When a response comes then the correlation IDs must be compared. The response is acknowledged and the user can enter a new message with BasicPublish.

Let's look at the receiver now. Add a new console project called RabbitMq.Rpc.Receiver and add the usual RabbitMq client library to it from NuGet. Here's the code for the consumer's Program.cs:

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

namespace RabbitMq.Rpc.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);

            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 = Console.ReadLine();
                IBasicProperties replyBasicProperties = channel.CreateBasicProperties();
                replyBasicProperties.CorrelationId = basicDeliveryEventArgs.BasicProperties.Correlati
                byte[] responseBytes = Encoding.UTF8.GetBytes(response);
                channel.BasicPublish("", basicDeliveryEventArgs.BasicProperties.ReplyTo, replyBasicPr
            };

            channel.BasicConsume("mycompany.queues.rpc", false, eventingBasicConsumer);
        }
    }
}
```

Most of this is standard code we've seen before. The main difference is within the implemented delegate body where the sender doesn't just acknowledge the message like before but also uses the ReplyTo property of the delivery arguments to publish a response. It also assigns the same correlation ID to its message as the one received in the initial message from the publisher.

Let's run this by starting the publisher first. Code execution will stop at the point where the user needs to enter an initial message. At this point start the RabbitMq.Rpc.Receiver application as well. Then send a message using the publisher's command window. The message will appear on the sender's screen. Use that window to send a response. The process continues with the publisher sending a message and the sender sending a response:


file:///C:/TestProjects/RabbitMq/Code/RabbitMqNetTests/RabbitMqNetTests/bin/Debug/RabbitMqNetTests.EXE

```
Enter your message and press Enter.
hello
Response: hi
Enter your message and press Enter.
how are you
Response: i'm fine and you?
Enter your message and press Enter.
i'm great. What are you doing tonight?
Response: nothing special, shall we watch a movie?
Enter your message and press Enter.
sure!
Response: ok, good
Enter your message and press Enter.
```

file:///C:/TestProjects/RabbitMq/Code/RabbitMqNetTests/RabbitMq.Rpc.Receiver/bin/Debug/RabbitMq.Rpc.Receiver.EXE

```
Message: hello  Enter your response:
hi
Message: how are you  Enter your response:
i'm fine and you?
Message: i'm great. What are you doing tonight?  Enter your response:
nothing special, shall we watch a movie?
Message: sure!  Enter your response:
ok, good
```

(https://dotnetcodr.files.wordpress.com/2016/08/rpc-messaging-demo-in-rabbitmq.png)

We have built a rudimentary chat application using RabbitMq.

In the next part (https://dotnetcodr.com/2016/08/25/messaging-with-rabbitmq-and-net-review-part-8-routing-and-topics/) we'll take up routing and topics.

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

FILED UNDER .NET, MESSAGING TAGGED WITH C#, MESSAGING, RABBITMQ

About Andras Nemes

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