RabbitMQ tutorial - Remote procedure call (RPC)

Remote procedure call (RPC)

(using the .NET client)

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.

In the second tutorial we learned how to use *Work Queues* to distribute time-consuming tasks among multiple workers.

But what if we need to run a function on a remote computer and wait for the result? Well, that's a different story. This pattern is commonly known as *Remote Procedure Call* or *RPC*.

In this tutorial we're going to use RabbitMQ to build an RPC system: a client and a scalable RPC server. As we don't have any time-consuming tasks that are worth distributing, we're going to create a dummy RPC service that returns Fibonacci numbers.

Client interface

To illustrate how an RPC service could be used we're going to create a simple client class. It's going to expose a method named `CallAsync` which sends an RPC request and blocks until the answer is received:

```
using var rpcClient = new RpcClient();

Console.WriteLine(" [x] Requesting fib({0})", n);
var response = await rpcClient.CallAsync(n);
Console.WriteLine(" [.] Got '{0}'", response);
```

> **A note on RPC #**
>
> Although RPC is a pretty common pattern in computing, it's often criticised. The problems arise when a programmer is not aware whether a function call is local or if it's a slow RPC. Confusions like that result in an unpredictable system and adds unnecessary complexity to debugging. Instead of simplifying software, misused RPC can result in unmaintainable spaghetti code.
>
> Bearing that in mind, consider the following advice:
>
> - Make sure it's obvious which function call is local and which is remote.
> - Document your system. Make the dependencies between components clear.
> - Handle error cases. How should the client react when the RPC server is down for a long time?
>
> When in doubt avoid RPC. If you can, you should use an asynchronous pipeline - instead of RPC-like blocking, results are asynchronously pushed to a next computation stage.

Callback queue

In general doing RPC over RabbitMQ is easy. A client sends a request message and a server replies with a response message. In order to receive a response we need to send a 'callback' queue address with the request:

```
var props = channel.CreateBasicProperties();
props.ReplyTo = replyQueueName;

var messageBytes = Encoding.UTF8.GetBytes(message);
```

```
channel.BasicPublish(exchange: string.Empty,
                     routingKey: "rpc_queue",
                     basicProperties: props,
                     body: messageBytes);

// ... then code to read a response message from the callback_queue ...
```

> **Message properties**
>
> The AMQP 0-9-1 protocol predefines a set of 14 properties that go with a message. Most of the properties are rarely used, with the exception of the following:
>
> - `Persistent`: Marks a message as persistent (with a value of `true`) or transient (any other value). Take a look at the second tutorial.
> - `DeliveryMode`: those familiar with the protocol may choose to use this property instead of `Persistent`. They control the same thing.
> - `ContentType`: Used to describe the mime-type of the encoding. For example for the often used JSON encoding it is a good practice to set this property to: `application/json`.
> - `ReplyTo`: Commonly used to name a callback queue.
> - `CorrelationId`: Useful to correlate RPC responses with requests.

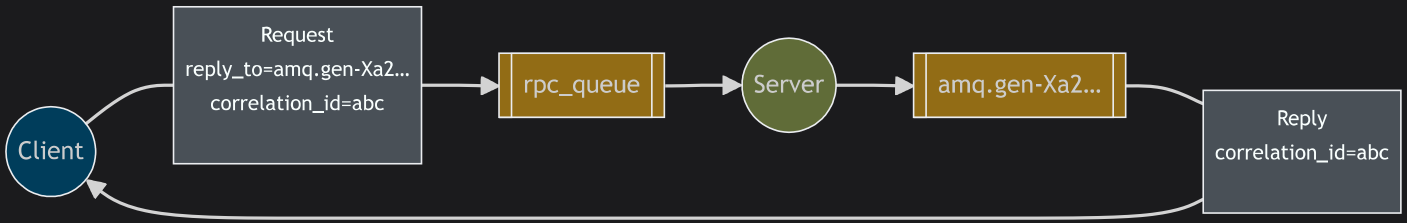
Correlation Id

In the method presented above we suggest creating a callback queue for every RPC request. That's pretty inefficient, but fortunately there is a better way - let's create a single callback queue per client.

That raises a new issue, having received a response in that queue it's not clear to which request the response belongs. That's when the `CorrelationId` property is used. We're going to set it to a unique value for every request. Later, when we receive a message in the callback queue we'll look at this property, and based on that we'll be able to match a response with a request. If we see an unknown `CorrelationId` value, we may safely discard the message - it doesn't belong to our requests.

You may ask, why should we ignore unknown messages in the callback queue, rather than failing with an error? It's due to a possibility of a race condition on the server side. Although unlikely, it is possible that the RPC server will die just after sending us the answer, but before sending an acknowledgment message for the request. If that happens, the restarted RPC server will process the request again. That's why on the client we must handle the duplicate responses gracefully, and the RPC should ideally be idempotent.

Summary



Our RPC will work like this:

- When the Client starts up, it creates an anonymous exclusive callback queue.
- For an RPC request, the Client sends a message with two properties: `ReplyTo`, which is set to the callback queue and `CorrelationId`, which is set to a unique value for every request.
- The request is sent to an `rpc_queue` queue.
- The RPC worker (aka: server) is waiting for requests on that queue. When a request appears, it does the job and sends a message with the result back to the Client, using the queue from the `ReplyTo` property.
- The client waits for data on the callback queue. When a message appears, it checks the `CorrelationId` property. If it matches the value from the request it returns the response to the application.

Putting it all together

The Fibonacci task:

```
static int Fib(int n)
{
    if (n is 0 or 1)
    {
        return n;
    }
    return Fib(n - 1) + Fib(n - 2);
}
```

We declare our fibonacci function. It assumes only valid positive integer input. (Don't expect this one to work for big numbers, and it's probably the slowest recursive implementation possible).

The code for our RPC server RPCServer.cs looks like this:

```
using System.Text;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;

var factory = new ConnectionFactory { HostName = "localhost" };
using var connection = factory.CreateConnection();
using var channel = connection.CreateModel();

channel.QueueDeclare(queue: "rpc_queue",
                     durable: false,
                     exclusive: false,
                     autoDelete: false,
                     arguments: null);
channel.BasicQos(prefetchSize: 0, prefetchCount: 1, global: false);
var consumer = new EventingBasicConsumer(channel);
channel.BasicConsume(queue: "rpc_queue",
                     autoAck: false,
                     consumer: consumer);
Console.WriteLine(" [x] Awaiting RPC requests");

consumer.Received += (model, ea) =>
{
    string response = string.Empty;

    var body = ea.Body.ToArray();
    var props = ea.BasicProperties;
    var replyProps = channel.CreateBasicProperties();
    replyProps.CorrelationId = props.CorrelationId;

    try
    {
        var message = Encoding.UTF8.GetString(body);
        int n = int.Parse(message);
        Console.WriteLine($" [.] Fib({message})");
```

```
            response = Fib(n).ToString();
        }
        catch (Exception e)
        {
            Console.WriteLine($" [.] {e.Message}");
            response = string.Empty;
        }
        finally
        {
            var responseBytes = Encoding.UTF8.GetBytes(response);
            channel.BasicPublish(exchange: string.Empty,
                                 routingKey: props.ReplyTo,
                                 basicProperties: replyProps,
                                 body: responseBytes);
            channel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false);
        }
    };

    Console.WriteLine(" Press [enter] to exit.");
    Console.ReadLine();

    // Assumes only valid positive integer input.
    // Don't expect this one to work for big numbers, and it's probably the slowest
    recursive implementation possible.
    static int Fib(int n)
    {
        if (n is 0 or 1)
        {
            return n;
        }

        return Fib(n - 1) + Fib(n - 2);
    }
```

The server code is rather straightforward:

- As usual we start by establishing the connection, channel and declaring the queue.
- We might want to run more than one server process. In order to spread the load equally over multiple servers we need to set the `prefetchCount` setting in `channel.BasicQos`.
- We use `BasicConsume` to access the queue. Then we register a delivery handler in which we do the work and send the response back.

The code for our RPC client RPCClient.cs:

```
using System.Collections.Concurrent;
using System.Text;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;

public class RpcClient : IDisposable
{
    private const string QUEUE_NAME = "rpc_queue";

    private readonly IConnection connection;
    private readonly IModel channel;
    private readonly string replyQueueName;
    private readonly ConcurrentDictionary<string, TaskCompletionSource<string>>
callbackMapper = new();

    public RpcClient()
    {
        var factory = new ConnectionFactory { HostName = "localhost" };

        connection = factory.CreateConnection();
        channel = connection.CreateModel();
        // declare a server-named queue
        replyQueueName = channel.QueueDeclare().QueueName;
        var consumer = new EventingBasicConsumer(channel);
        consumer.Received += (model, ea) =>
        {
            if (!callbackMapper.TryRemove(ea.BasicProperties.CorrelationId, out
var tcs))
                return;
            var body = ea.Body.ToArray();
            var response = Encoding.UTF8.GetString(body);
            tcs.TrySetResult(response);
        };

        channel.BasicConsume(consumer: consumer,
                             queue: replyQueueName,
                             autoAck: true);
    }

    public Task<string> CallAsync(string message, CancellationToken
cancellationToken = default)
```

```
    {
        IBasicProperties props = channel.CreateBasicProperties();
        var correlationId = Guid.NewGuid().ToString();
        props.CorrelationId = correlationId;
        props.ReplyTo = replyQueueName;
        var messageBytes = Encoding.UTF8.GetBytes(message);
        var tcs = new TaskCompletionSource<string>();
        callbackMapper.TryAdd(correlationId, tcs);

        channel.BasicPublish(exchange: string.Empty,
                             routingKey: QUEUE_NAME,
                             basicProperties: props,
                             body: messageBytes);

        cancellationToken.Register(() => callbackMapper.TryRemove(correlationId,
out _));
        return tcs.Task;
    }

    public void Dispose()
    {
        // closing a connection will also close all channels on it
        connection.Close();
    }
}

public class Rpc
{
    public static async Task Main(string[] args)
    {
        Console.WriteLine("RPC Client");
        string n = args.Length > 0 ? args[0] : "30";
        await InvokeAsync(n);

        Console.WriteLine(" Press [enter] to exit.");
        Console.ReadLine();
    }

    private static async Task InvokeAsync(string n)
    {
        using var rpcClient = new RpcClient();

        Console.WriteLine(" [x] Requesting fib({0})", n);
        var response = await rpcClient.CallAsync(n);
        Console.WriteLine(" [.] Got '{0}'", response);
    }
}
```

The client code is slightly more involved:

- We establish a connection and channel and declare an exclusive 'callback' queue for replies.
- We subscribe to the 'callback' queue, so that we can receive RPC responses.
- Our `Call` method makes the actual RPC request.
- Here, we first generate a unique `CorrelationId` number and save it to identify the appropriate response when it arrives.
- Next, we publish the request message, with two properties: `ReplyTo` and `CorrelationId`.
- At this point we can sit back and wait until the proper response arrives.
- For every response message the client checks if the `CorrelationId` is the one we're looking for. If so, it saves the response.
- Finally we return the response back to the user.

Making the Client request:

```
using var rpcClient = new RpcClient();

Console.WriteLine(" [x] Requesting fib({0})", n);
var response = await rpcClient.CallAsync(n);
Console.WriteLine(" [.] Got '{0}'", response);
```

Now is a good time to take a look at our full example source code (which includes basic exception handling) for RPCClient.cs and RPCServer.cs.

Set up as usual (see tutorial one):

Our RPC service is now ready. We can start the server:

```
cd RPCServer
dotnet run
# => [x] Awaiting RPC requests
```

To request a fibonacci number run the client:

```
cd RPCClient
dotnet run
# => [x] Requesting fib(30)
```

The design presented here is not the only possible implementation of a RPC service, but it has some important advantages:

- If the RPC server is too slow, you can scale up by just running another one. Try running a second `RPCServer` in a new console.
- On the client side, the RPC requires sending and receiving only one message. No synchronous calls like `QueueDeclare` are required. As a result the RPC client needs only one network round trip for a single RPC request.

Our code is still pretty simplistic and doesn't try to solve more complex (but important) problems, like:

- How should the client react if there are no servers running?
- Should a client have some kind of timeout for the RPC?
- If the server malfunctions and raises an exception, should it be forwarded to the client?
- Protecting against invalid incoming messages (eg checking bounds, type) before processing.

> If you want to experiment, you may find the management UI useful for viewing the queues.