INTERPROZESSKOMMUNIKATION MIT C#

Mächtig und praxiserprobt

Mit RabbitMQ können verschiedene Prozesse asynchron miteinander kommunizieren.

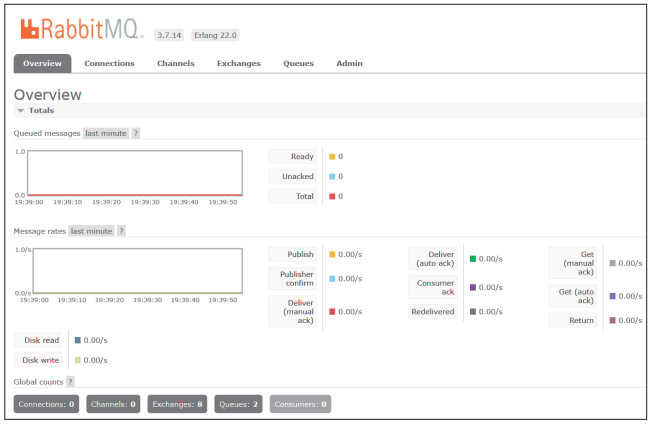
Sie kennen das sicherlich: Die Anwendung wächst und gedeiht, und plötzlich sind Sie mit dem Problem konfrontiert, die Komponenten entkoppeln zu wollen, etwa um das kostspielige Erstellen von Berichten in einen anderen Prozess auszulagern. Natürlich soll aber nach bestem Wissen und Gewissen die Koppelung zwischen den Prozessen gering gehalten werden. In solchen Fällen bietet sich der Einsatz einer Software wie RabbitMQ an.

Bei RabbitMQ handelt es sich um einen sogenannten Message Broker, der als Open-Source-Software zur Verfügung steht. Er basiert auf dem Warteschlangenprinzip und erlaubt das Verknüpfen von zwei oder mehr Prozessen, die nicht über eine Implementierung desselben Kommunikationsprotokolls verfügen müssen [1]. Und das funktioniert so: Ein Anbieter (Publisher) möchte eine Aufgabe an einen anderen Prozess abgeben und sendet dazu eine Nachricht über den Message Broker. Dieser reiht diese Nachricht in die Warteschlange ein und verteilt diese anhand eines Routings an andere Prozesse, die sich als Abonnenten (Subscriber) beim Broker angemeldet haben und die mit der Nachricht verknüpfte Aufgabe entgegennehmen, sobald sie ausreichend Kapazitäten verfügbar haben. Ist die Aufgabe erledigt, kann der Abonnent auf die Nachricht antworten, sodass der Publisher mit den Ergebnissen weiterarbeiten kann.

Dabei wird eine erhöhte Parallelität bei der Aufgabenerfüllung angestrebt: Statt eine Aufgabe selbst zu erledigen und damit den eigenen Prozess zu belasten, kann ein Worker-Prozess die Aufgabe übernehmen, der nicht einmal auf demselben Rechner laufen muss.

Ein besonders mächtiges Merkmal ist das schon genannte Routing: Anhand verschiedener Kriterien kann angegeben werden, an welche Empfänger bestimmte Formen von Nachrichten zuzustellen sind. So lässt sich beispielsweise eine Nachricht an alle Warteschlangen senden, nur an eine bestimmte oder auch mithilfe von Wildcards an eine Auswahl von Warteschlangen.

Eine der Stärken des Brokers ist die Möglichkeit, Ausfallsicherheit zu gewährleisten. Einzelne Nodes lassen sich zu



Die Verwaltungskonsole das RabbitMQ-Servers (Bild 1)

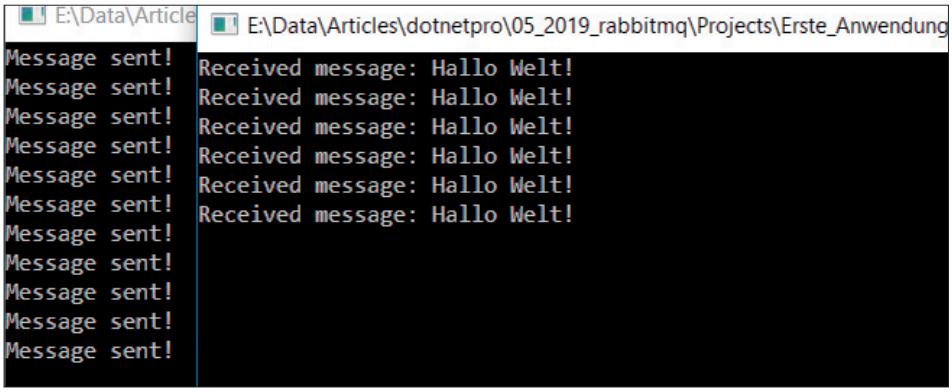
Listing 1: Der Publisher

```
static void Main(string[] args)
{
    var factory = new ConnectionFactory();
    factory.HostName = "localhost";

    using (var con = factory.CreateConnection())
    {
        using (var channel = con.CreateModel())
        {
            var basicProperties =
                channel.CreateBasicProperties();
            basicProperties.Persistent = true;

            channel.QueueDeclare(queue: "hello",
                durable: false,
                exclusive: false,
                autoDelete: false,
                arguments: null);

            var msg = "Hallo Welt!";
            var body = Encoding.UTF8.GetBytes(msg);

            while (Console.ReadKey().Key ==
                ConsoleKey.Enter)
            {
                channel.BasicPublish(exchange: "",
                    routingKey: "hello",
                    basicProperties:
                        basicProperties,
                    body: body);
                    Console.WriteLine(
                        "Message sent!");
            }
        }
    }
}
```

einem Cluster zusammenfassen, die Queues werden dabei auf allen Nodes gespiegelt. Dass hierbei keine Nachrichten verloren gehen, dafür sorgt die Software mit vielen Mitteln.

Bestätigen oder verwerfen

Für jede Nachricht, die über RabbitMQ verschickt wird, kann angegeben werden, ob der Publisher über den Empfang informiert werden soll oder nicht. So kann dem Anwender beispielsweise wie beim Videoportal YouTube direkt mitgeteilt werden, ob sein Video-Upload angenommen wurde, während die eigentliche Verarbeitung des Videos noch im Hintergrund weiterläuft. Ist das Processing abgeschlossen, kann der Worker etwa den Publisher benachrichtigen, damit dieser die Information über die erfolgte Verarbeitung an den Nutzer weitergibt; dieser muss dann nicht unnötige Zeit auf der Website verharren.



Das Zusammenspiel zwischen Publisher und Subscriber (Bild 2)

Listing 2: Der Subscriber

```
static void Main(static void Main(string[] args)
{
    var factory = new ConnectionFactory();
    factory.HostName = "localhost";

    using (var con = factory.CreateConnection())
    {
        using (var channel = con.CreateModel())
        {
            channel.QueueDeclare(queue: "hello",
                durable: false,
                exclusive: false,
                autoDelete: false,
                arguments: null);

            var consumer =
                new EventingBasicConsumer(channel);
            consumer.Received += Consumer_Received;

            channel.BasicConsume(queue: "hello",
                autoAck: true,
                consumer: consumer);

            Console.ReadLine();
        }
    }

    Console.ReadLine();
}

private static void Consumer_Received(object
    sender, BasicDeliverEventArgs e)
{
    var body = e.Body;
    var message = Encoding.UTF8.GetString(body);
    Console.WriteLine("Received message: {0}",
        message);
}
```

Das Protokoll, das RabbitMQ zur Kommunikation zwischen Anbieter und Abonnent zugrunde liegt, nennt sich AMQP (Advanced Message Queuing Protocol). Dabei handelt es sich um einen offenen Standard für nachrichtenorientierte Software-Middleware, der ursprünglich vor allem für die Bankenindustrie entwickelt wurde und eine hohe Zuverlässigkeit gewährleisten soll [2].

RabbitMQ installieren

Die Installation des RabbitMQ-Servers erfolgt unter Windows über einen separaten Installer, der eine Erlang-Installation benötigt [3]. In der Grundkonfiguration ist der Server bereits für den Zweck dieses Artikels einsatzbereit. Navigieren Sie nach dem Setup in der Kommandozeile in das *sbin*-Verzeichnis der Installation und führen Sie folgendes Kommando aus:

```
bbitmq-plugins enable rabbitmq_management
```

Starten Sie den Server dann neu. Anschließend ist eine Verwaltungsoberfläche im Webbrowser über die Webadresse http://localhost:15672 mit dem Standard-Login *guest/guest* erreichbar. Sie hält in einer Übersicht die verschickten Nachrichten, Verbindungen, Verbindungskanäle, Warteschlangen und mehr bereit (Bild 1) [4].

Die erste Nachricht

Das erste Beispiel verwendet die .NET-Client-Bibliotheken von RabbitMQ, um zwei Anwendungen zu entwickeln, die miteinander über den Message Broker kommunizieren. Publisher und Abonnent sind Konsolenanwendungen. In beiden Projektmappen wird mithilfe von NuGet das Package *RabbitMQ.Client* in der Version 5.1.0 installiert:

```
Install-Package RabbitMQ.Client -Version 5.1.0
```

Im Publisher baut das *ConnectionFactory*-Objekt des RabbitMQ-Clients eine Verbindung zum Broker auf und erzeugt einen Kanal, der die Brücke zum RabbitMQ-API schlägt (Listing 1). Nach erfolgreicher Initialisierung benötigt die App noch eine Warteschlange, über welche die Nachrichten geschickt werden. Diese wird erzeugt, sofern sie noch nicht existiert, und mittels des *durable*-Flags dauerhaft gespeichert, sodass sie auch nach Neustart des RabbitMQ-Servers bestehen bleibt

Auch die Nachrichten sollen nach einem Neustart verfügbar bleiben. Hierzu wird eine Instanz der Schnittstelle *IBasicProperties* initialisiert, die als Container für zusätzliche Header-Informationen der Nachricht dient und über die Methode *CreateBasicProperties()* erzeugt wird. Bei diesem Objekt wird das *Persistent*-Flag auf *true* gesetzt. Die Nachrichten sind nun auch nach einem Neustart verfügbar.

Der letzte Teil des Listings implementiert eine Tastaturabfrage: Solange der Nutzer die [Enter]-Taste drückt, wird die Byte-Repräsentation der Nachricht an den Broker gesendet, der diese dann an den Abonnenten weiterleiten soll.

Der Subscriber baut nach demselben Prinzip eine Verbindung auf und wartet auf derselben Warteschlange auf Nachrichten (Listing 2). Dies übernimmt ein Eventhandler, der die Byte-Repräsentation einer erhaltenen Nachricht in einen String verwandelt, um diese schließlich wieder auf der Konsole auszugeben. Der Aufruf von *BasicConsume()* veranlasst den Abonnenten, auf eingehende Nachrichten in der Warteschlange zu hören und diese bei Ankunft zu konsumieren.

Listing 3: Verbindung mit einem Worker erkennen

```
var factory = new ConnectionFactory();
factory.HostName = "localhost";

using (var con = factory.CreateConnection())
{
    using (var channel = con.CreateModel())
    {
        channel.QueueDeclare(queue: "worker",
            durable: false,
            exclusive: false,
            autoDelete: false,
            arguments: null);

        var consumer = new EventingBasicConsumer(
            channel);
        consumer.Received += (sender, e) =>
        {
            var body = e.Body;
            var basicProperties = e.BasicProperties;
            var replyProperties =
                channel.CreateBasicProperties();
            replyProperties.CorrelationId =
                basicProperties.CorrelationId;

            var url = Encoding.UTF8.GetString(body);
            Console.WriteLine(
                "Processing url: {0}", url);

            string content = "";

            try
            {
                content = ProcessUrl(url);
            }
            catch (Exception)
            {
                Console.WriteLine(
                    "failed to read url {0}", url);
            }
            finally
            {
                var response =
                    Encoding.UTF8.GetBytes(content);
                channel.BasicPublish(exchange: "",
                    routingKey: basicProperties.
                    ReplyTo, basicProperties:
                    replyProperties, body: response);
                channel.BasicAck(deliveryTag:
                    e.DeliveryTag, multiple: false);
            }
        };

        channel.BasicConsume(queue: "worker",
            autoAck: false,
            consumer: consumer);

            Console.ReadLine();
        }
    }

    Console.ReadLine();
}

private static string ProcessUrl(string url)
{
    string content = "";
    using (WebClient client = new WebClient())
    {
        content = client.DownloadString(url);
    }

    return content;
}
```

Starten Sie anschließend zunächst den Publisher und dann den Subscriber, so wird die Queue im Hintergrund angelegt und Nachrichten an sie zugestellt. In der Verwaltungsoberfläche können Sie sich einen Überblick über die verschickten Daten und verschiedene Auswertungen verschaffen. Die Nachrichten werden nun im Subscriber-Prozess zugestellt und auf der Konsole ausgegeben (Bild 2).

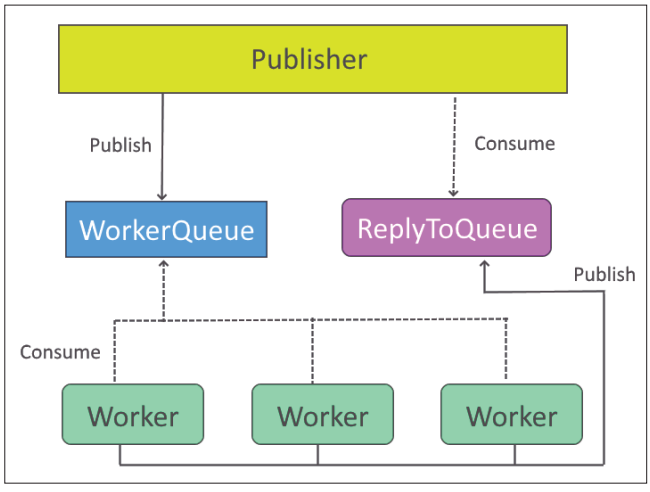
Außerdem werden die Nachrichten, die der Publisher verschickt, die aber derzeit nicht durch einen Subscriber-Prozess empfangen werden können, vorrätig gehalten und beim erneuten Start des Subscribers automatisch zugestellt.

Arbeit verteilen

Im Folgenden soll eine Architektur entwickelt werden, in der mehrere Worker-Prozesse laufen, die eine Aufgabe entgegennehmen, wobei das Prinzip gilt „Der Erste gewinnt“: Der erste Prozess, der die Aufgabe entgegennimmt, ist verantwortlich für die Umsetzung. Dabei ist vor allem die Frage interessant, was passiert, wenn einer der Worker nicht mehr verfügbar ist, also seine Aufgabe nicht erledigen kann. Im aktuellen Zustand ginge die Nachricht verloren, da sie automatisch aus der Queue gelöscht werden würde, sobald diese zugestellt würde.

Im vorliegenden Fall soll die Nachricht erneut an einen anderen Worker gesendet werden. Dabei soll RabbitMQ automatisch erkennen, ob die Verbindung mit einem Worker beendet wurde, und entsprechend handeln. Hierzu deaktiviert der Subscriber das automatische Quittieren, indem in Listing 3 in der Methode *BasicConsume()* der Parameter *autoAck: false* übergeben wird. Würde dieses Flag weiter aktiv bleiben, so würden Nachrichten stets nach der Auslieferung als erfolgreich übermittelt erscheinen, unabhängig davon, ob tatsächlich ein Abonnent darauf reagiert hat.

Anschließend wird eine Quittung gesendet, falls die Nachricht verarbeitet wurde. Falls diese nicht bei RabbitMQ ankommt, so wird die Nachricht automatisch erneut an die Warteschlange zugestellt. In dem Beispiel lädt dann ein *WebClient*-Objekt den Inhalt hinter einem URL herunter und gibt sie an den Publisher zurück.



Die Architektur der Beispielanwendung (Bild 3)

Listing 4: Umgebauter Publisher

```
var replyQueue = channel.QueueDeclare().QueueName;
var basicProperties =
    channel.CreateBasicProperties();
basicProperties.Persistent = true;
basicProperties.ReplyTo = replyQueue;
basicProperties.CorrelationId =
    Guid.NewGuid().ToString();

channel.QueueDeclare(queue: "worker",
    durable: false,
    exclusive: false,
    autoDelete: false,
    arguments: null);

var url = "http://www.dotnetpro.de";
var body = Encoding.UTF8.GetBytes(url);

var consumer = new EventingBasicConsumer(channel);
consumer.Received += (m, e) =>
{
    var response = e.Body;
    var content = Encoding.UTF8.GetString(response);
    Console.WriteLine(content);
};
channel.BasicConsume(queue: replyQueue,
    consumer: consumer, : true);

while (Console.ReadKey().Key == ConsoleKey.Enter)
{
    channel.BasicPublish(exchange: "",
        routingKey: "worker",
        basicProperties: basicProperties,
        body: body);
    Console.WriteLine("Work published.");
}
```

Um nun auf die erfolgreiche Arbeit reagieren zu können, muss eine zweiseitige Kommunikation zwischen Publisher und Worker bestehen. Hierzu wird die Nachricht beim Versand durch den Publisher mit einem *ReplyTo*-Merker versehen. Zusätzlich muss der Publisher aus einer speziellen *ReplyTo*-Queue konsumieren. Bei erfolgreicher Abarbeitung kann der Worker nun eine Nachricht in diese Queue geben und den Publisher über die erfolgreiche Abarbeitung informieren. Bild 3 veranschaulicht die komplette Architektur. Der Publisher sieht nun aus wie in Listing 4.

Als Queue für die bidirektionale Kommunikation dient nun eine neue Queue, die zur Laufzeit des Programms besteht. Die Angaben zu den Kommunikationsmöglichkeiten über diese Queue werden als eine Art Umschlag an jede Nachricht gehängt und zeigen dem Worker, wie er auf die Nachricht zu reagieren hat. Der Publisher selbst hört auf diese Queue ▶

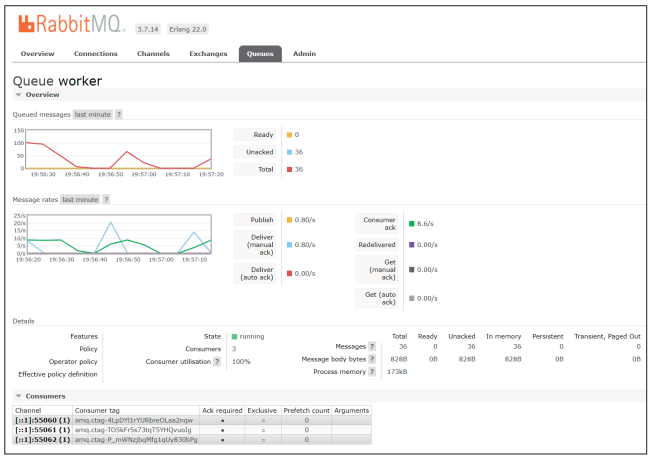
Das Zusammenspiel von mehreren Workern mit dem Publisher (Bild 4)

und wird benachrichtigt, sobald die Arbeit erfolgt ist. Um in weiteren Anwendungsfällen die ursprüngliche Nachricht selbst zu identifizieren, wird eine Korrelations-ID generiert.

Auch der Worker im Beispiel hat sich ein wenig verändert. Die Antwort erfolgt nun in die Queue, die der Publisher als *ReplyTo*-Queue festgelegt hat. Starten Sie anschließend den Publisher erneut und dazu mehrere Instanzen des Subscriber-Prozesses. Wenn Sie nun eine Nachricht versenden, so sehen Sie, wie der gerade verfügbare Subscriber diese entgegennimmt und nach einer kurzen Arbeitszeit eine Antwort an den Publisher schickt, der diese dann ausgibt (Bild 4).

Dies ist das grundlegende Prozedere bei der Verteilung von Arbeit. Als Grundlage kommt hier das Entwurfsmuster Competing Consumers zum Einsatz: RabbitMQ entscheidet, welcher Abonnent die Nachricht erhält, allerdings konkurrieren die Subscriber untereinander [5]. In der Standardkonfiguration sendet RabbitMQ die Nachricht per Rundlauf (Round Robin) an den nächsten verfügbaren Subscriber.

Werfen Sie auch einmal einen Blick auf den Durchsatz in den Queues selbst. Die Verwaltungskonsole gibt hier einen guten Einblick in die Kommunikation zwischen den Prozessen (Bild 5). Hier zeigt sich zum Beispiel, wie viele Nachrichten sich zurückstauen, da kein Worker bereit ist, die offenen Aufgaben zu übernehmen.



Übersicht der Reply-Queue (Bild 5)

Weiterer Ausblick

RabbitMQ ist ein sehr mächtiges Werkzeug, das verschiedene Prozesse verwaltet, um sie asynchron miteinander kommunizieren zu lassen. Dabei ist es vielseitig, praxiserprobt und es ermöglicht ein automatisches Routing von Nachrichten je nach Anwendungsfall.

Im produktiven Einsatz kommen etwa noch die Erweiterbarkeit und die Authentifizierung auf verschiedene Arten wie LDAP oder mit Zertifikaten hinzu. Mehrere Server-Knoten lassen sich relativ unproblematisch zusammenschließen und kontrollieren.

Diese kleine Einführung hat Ihnen die Verwendung von RabbitMQ hoffentlich etwas schmackhafter gemacht und dazu ermuntert, den Einsatz in eigenen Projekten in Erwägung zu ziehen. Für die weitere Einarbeitung empfiehlt sich die hervorragende Dokumentation von RabbitMQ, die all das noch genauer erläutert [6]. ■

[1] RabbitMQ Features,
https://www.rabbitmq.com/#features
[2] Wikipedia, Advanced Message Queuing Protocol,
www.dotnetpro.de/SL1911RabbitMQ1
[3] Downloading and Installing RabbitMQ,
www.dotnetpro.de/SL1911RabbitMQ2
[4] RabbitMQ, Management Plugin,
www.dotnetpro.de/SL1911RabbitMQ3
[5] Competing Consumers,
www.dotnetpro.de/SL1911RabbitMQ4
[6] RabbitMQ, Documentation,
www.dotnetpro.de/SL1911RabbitMQ5

Thomas Symalla
begann mit dem Programmieren auf einem CPC464. Heute ist er Softwareentwickler im Automotive-Bereich und studiert Informatik mit dem Schwerpunkt auf Computergrafik und Computer Vision. Sie erreichen ihn über **hallo@thomassymalla.de**

dnpCode A1911RabbitMQ