



Neue Ansätze für asynchrone Kommunikation

Die Zukunft des Messaging

Kaum ein Bereich ist so stabil wie Messaging – seit 2002 hat sich der einschlägige JMS-Standard nicht weiterentwickelt. Aber die Welt dreht sich weiter: Außerhalb der Java-Welt wird AMQP [1] als Messaging-Lösung immer wichtiger. Jetzt steht mit Spring AMQP ein Projekt bereit, das diesen Standard auch im Java-Bereich unterstützt. Zeit also, einen Blick zu wagen.

von Eberhard Wolff

Die Kommunikation zwischen verteilten Systemen findet oft mit RPC (Remote Procedure Call) statt, bei denen Methoden auf dem Server aufgerufen werden. In diese Kategorie fallen Technologien wie RMI (Remote Method Invocation), SOAP Web Services oder CORBA. Im Wesentlichen wird eine Methode in einem entfernten Prozess aufgerufen und man wartet auf das Ergebnis. Dieses Vorgehen schafft einige Probleme: So muss sich der Entwickler mit Netzwerkproblemen oder -ausfällen beschäftigen und auch mit langen Antwortzeiten zurecht kommen. Zudem gibt der Methodenaufruf genau an, was passieren soll, also beispielsweise: „Lege eine neue Bestellung in der Datenbank an“.

Bei einer Nachricht wie „Neue Bestellung“ kann der Server entscheiden, was er aufgrund dieser Nachricht tut. Er kann die Bestellung in der Datenbank anlegen oder die Rechnung vorbereiten. Dadurch sind die Systeme

stärker entkoppelt, da jedes System selber entscheiden kann, wie es auf eine Nachricht reagiert. Außerdem kann die Bearbeitung der Nachricht zu einem anderen Zeitpunkt stattfinden als das Verschicken, sodass Sender und Empfänger auch zeitlich entkoppelt sind.

Bei einem Netzwerkausfall können Nachrichten lokal gespeichert und die Übertragung so lange verzögert werden, bis das Netzwerk wieder verfügbar ist. Bei einem Fehler in der Bearbeitung der Nachricht kann diese dann so oft neu übertragen werden, bis sie erfolgreich bearbeitet worden ist. Typische Probleme verteilter Systeme wie Netzwerkausfälle oder Latenzzeiten können also mit Messaging elegant gelöst werden.

Dieser Vorteil bedeutet aber auch, dass man die Architektur anpassen muss. Das System muss asynchron arbeiten, was ein anderes Denken als die synchrone Abarbeitung von Methoden erfordert. Dieses Vorgehen ist jedoch nicht so ungewöhnlich: Ansätze wie AJAX (Asynchronous JavaScript and XML) sind ebenfalls

asynchron, sodass eine solche Architektur auch zum Beispiel im Bereich GUI relevant ist. Zum tieferen Einstieg in asynchrone Mechanismen lohnt sich [2].

AMQP: Der neue Messaging-Standard

AMQP (Advanced Message Queuing Protocol) ist ein Standard im Messaging-Bereich. Er definiert nicht etwa ein API, sondern ein Netzwerkprotokoll, das von Messaging-Lösungen zum Übertragen der Nachrichten genutzt wird. Die Standardisierung auf dieser Ebene hat einige Vorteile:

- Ein Client benötigt nur eine Library, auch wenn er verschiedene AMQP-Implementierungen parallel nutzt.
- Die Abhängigkeit zu einem Hersteller verringert sich, weil Produkte einfach gegeneinander ausgetauscht werden können. Da sie alle dasselbe Netzwerkprotokoll verwenden, verhalten sie sich gleich. Natürlich sind dennoch in der Praxis Unterschiede bezüglich Performance oder Inkompatibilitäten nicht ausgeschlossen.
- Das Protokoll ist effizient. So werden beispielsweise Daten binär übertragen.
- Schon jetzt gibt es Unterstützung für praktisch alle wichtigen Programmiersprachen und Betriebssysteme.

RabbitMQ

RabbitMQ ist eine Implementierung des AMQP-Standards. Es implementiert neben AMQP 0.8, 0.9 und 0.9.1 auch noch zahlreiche weitere Messaging-Protokolle wie STOMP oder XMPP. Außerdem ist es auf der Amazon-EC2-Plattform sehr beliebt und auch die Basis für NASAs Nebula-Cloud-Initiative. RabbitMQ ist in der Programmiersprache Erlang implementiert und liegt im Moment in der Version 2.4.1 vor. Auch Enterprise-Features wie Clustering werden von RabbitMQ unterstützt.

Im Java-Bereich gibt es ebenfalls Support: Angeboten werden ein Grails-Plug-in, ein nativer Java-Client und auch Unterstützung für Scala/Lift. In diesem Artikel wird die Unterstützung von Java mithilfe von Spring AMQP 1.0.0RC1 [3] im Mittelpunkt stehen.

Wie geht nun die Kommunikation mit AMQP konkret vonstatten? Im einfachsten Fall schickt der Producer Nachrichten an eine Queue, die dann die Nachrichten speichert, bis sie vom Consumer abgeholt werden. Queues können „durable“ sein: Dann überstehen sie auch einen Server-Neustart. Sind sie „exclusive“, so stehen sie nur für eine Verbindung zur Verfügung, und Queues mit autoDelete werden gelöscht, wenn die Verbindung abgebaut wird. Queues werden meistens vom Consumer der Nachrichten erzeugt. Alle Ressourcen in AMQP sind dynamisch und werden zur Laufzeit erzeugt.

Anzeige

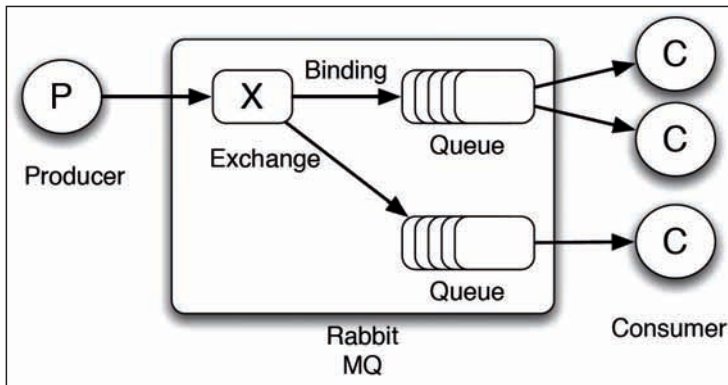


Abb. 1: Grundlegende Elemente von AMQP: Exchange, Queue und Binding

Listing 1 zeigt den dazu notwendigen Code: Zunächst wird eine *CachingConnectionFactory* angelegt, mit der auf den RabbitMQ-Server zugegriffen werden kann. Dann wird eine Instanz von *RabbitAdmin* erzeugt. Dieses Objekt erlaubt es, Ressourcen im RabbitMQ-Server anzulegen. Es implementiert das Interface *AmqpAdmin*, das die Operation des AMQP-Standards enthält. Dann wird eine Queue erzeugt oder, genauer gesagt, deklariert. Sollte sie nämlich schon vorhanden sein, wird sie

nicht etwa erneut erzeugt, sondern lediglich eine Referenz auf die Queue zurückgegeben, wenn sie der geforderten Konfiguration entspricht. Gibt es bereits eine Queue mit dem übergebenen Namen, aber einer anderen Konfiguration, wird eine Exception geworfen.

In der nächsten Zeile in Listing 1 wird ein *RabbitTemplate* erzeugt. Auch hier gibt es das RabbitMQ-spezifische Interface *RabbitOperations* und das allgemeinere AMQP-Interface *AmqpTemplate*. Man erkennt hier, dass Spring AMQP in Zukunft auch andere AMQP-Implementierungen neben RabbitMQ unterstützen könnte.

Mithilfe des *RabbitTemplates* wird dann eine Nachricht verschickt und anschließend empfangen. Nun fragt sich noch, was genau überhaupt übertragen wird: Der AMQP-Standard definiert nur binäre Nachrichten und Strings können auf unterschiedliche Art und Weise dargestellt werden. In Spring AMQP gibt es dafür *MessageConverter*: Sie definieren, wie eine Nutzlast für eine Nachricht kodiert wird. Wird nichts anderes angegeben, wird der *SimpleMessageConverter* genutzt. Strings werden dann mit einem konfigurierbaren Encoding kodiert, ein Byte-Array direkt übertragen und serialisierbare Objekte serialisiert. Alternativ ist es auch möglich, einen *JsonMessageConverter* zu nutzen. Dann werden die Ob-

Listing 1

```
ConnectionFactory conFactory = new CachingConnectionFactory("localhost");
RabbitAdmin admin = new RabbitAdmin(conFactory);
admin.declareQueue(new Queue("myQueue"));
RabbitTemplate template = new RabbitTemplate(conFactory);
template.convertAndSend("myQueue", "Hi AMQP!");
String receive = (String) template.receiveAndConvert("myQueue");
Assert.assertEquals("Hi AMQP!", receive);
```

Listing 2

```
Queue fanoutQueue = new Queue("fanoutQueue");
admin.declareQueue(fanoutQueue);

FanoutExchange fanoutExchange = new FanoutExchange("myFanout");
admin.declareExchange(fanoutExchange);

admin.declareBinding(
    BindingBuilder.from(fanoutQueue)
        .to(fanoutExchange));

template.setExchange("myFanout");
template.convertAndSend("Hi Fanout!");

String receive = (String) template.receiveAndConvert("fanoutQueue");
Assert.assertEquals("Hi Fanout!", receive);
```

Listing 3

```
Queue directQueue = new Queue("direct");
admin.declareQueue(directQueue);
admin.declareBinding(BindingBuilder
    .from(directQueue)
```

```
.to(new DirectExchange("amq.direct"))
    .with("hello.Key"));
template.setExchange("amq.direct");
template.convertAndSend("amq.direct", "drop.Me", "I will be dropped!");
template.convertAndSend("amq.direct", "hello.Key", "Hi Direct!");
Assert.assertEquals("Hi Direct!", template.receiveAndConvert("direct"));
Assert.assertNull(template.receiveAndConvert("direct"));
```

Listing 4

```
<bean id="connectionFactory"
    class="org.springframework.amqp.rabbit.connection.CachingConnectionFactory">
    <property name="username" value="guest"/>
    <property name="password" value="guest"/>
    <constructor-arg value="localhost" />
</bean>

<bean id="rabbitTemplate"
    class="org.springframework.amqp.rabbit.core.RabbitTemplate">
    <constructor-arg ref="connectionFactory" />
    <property name="routingKey" value="invoice.USD" />
</bean>

<rabbit:listener-container
    connection-factory="connectionFactory">
    <rabbit:listener ref="consumer" method="consume"
        queue-names="my.amqp.queue2" />
</rabbit:listener-container>
```

Listing 5

```
String response = (String)
    rabbitTemplate.convertSendAndReceive(
        "my.fanout", "", "test");
```

jekte als JSON (JavaScript Object Notation) übertragen, das mit der Java-Library Jackson erzeugt bzw. dekodiert wird. Eine andere Möglichkeit ist der *MarshallingMessageConverter*. Die Nachrichten enthalten dann XML, das mithilfe eines XML-Java-APIs erzeugt wird. Dabei wird das Spring-OXM-Mapping genutzt, das eine Abstraktion über verschiedene XML-Technologien wie JAXB, DOM oder SAX darstellt, sodass diese alle mit Spring AMQP genutzt werden können.

Eine Queue ist nicht genug

Wie erwähnt, reiht eine Queue Nachrichten in eine Warteschlange ein. Das ist für viele Szenarien nicht ausreichend, weil Nachrichten an unterschiedliche Consumer geleitet werden müssen. Daher gibt es in AMQP Exchanges. Diese routen Messages und sind zustandslos. In Listing 1 war kein Exchange angegeben. Daher wurde der DefaultExchange genutzt, der Nachrichten an jene Queue schickt, deren Name dem Routing Key entspricht. Durch die Kombination der beiden für sich sehr einfachen Bausteine „Queue“ und „Exchange“ können Konfigurationen erstellt werden, die auch komplexen Anforderungen gerecht werden. Das zeigt die Eleganz des AMQP-Ansatzes.

Abbildung 1 zeigt das Zusammenspiel: Eine Queue ist mit einem Exchange durch ein Binding verbunden. Der Producer schickt Nachrichten an ein Exchange. Dieser leitet sie an eine oder mehrere Queues weiter. Die Queues werden dann von den Consumern abgefragt, wobei eine Queue durchaus auch von mehreren Consumern genutzt werden kann.

Wie der Exchange mit Nachrichten umgeht, ergibt sich aus dem Typ. Einen eigenen Exchange-Typ zu implementieren ist möglich, aber meistens sind die vordefinierten ausreichend:

- Bei einem FanoutExchange werden die Nachrichten einfach an alle verbundenen Queues weitergeleitet.

Und was ist mit JMS?

JMS (Java Message Service) ist ein standardisiertes API für den Zugriff von Java auf Messaging-Systeme. Sie ist seit 2002 stabil in der Version 1.1. Gegenüber AMQP verfolgt JMS einen anderen Ansatz: JMS ist ein Standard für ein API. Das bei AMQP standardisierte Netzwerkprotokoll kann bei JMS beliebig sein. Daher bringt jede JMS-Implementierung auch ihre eigene Client Bibliothek mit. Außerdem ist JMS weniger flexibel, da die Aufteilung in Exchanges für Routing und Queues für das Speichern der Nachrichten fehlt. Stattdessen bietet JMS nur Lösungen für Point-to-Point- oder Publish/Subscribe-Szenarien mit. Zudem werden bei JMS Ressourcen oft statisch definiert, während AMQP grundsätzlich alles dynamisch konfiguriert.

Prinzipiell kann JMS als Schnittstelle für AMQP genutzt werden und im Rahmen der AMQP-Standardisierungen gibt es auch entsprechende Bestrebungen, die aber noch nicht zu einem endgültigen Ergebnis geführt haben [4]. Unter [5] gibt es schon ein passendes Projekt, um JMS über AMQP zu realisieren.

Übrigens gibt es im Spring Core Framework auch eine Unterstützung für JMS, die der hier vorgestellten AMQP-Unterstützung weitgehend entspricht. Wer sich also mit dem oft komplexen JMS-API nicht mehr beschäftigen will, sollte einen Blick darauf wagen.

Das ist also für Broadcast-Nachrichten hilfreich. Listing 2 zeigt den entsprechenden Code: Zunächst wird die Queue definiert, dann der Exchange und das Binding zwischen beiden. Im *RabbitTemplate* wird festgelegt, an welchen Exchange die Nachricht geschickt werden soll. Die Nachricht wird dann von dem FanoutExchange an alle Queues zugestellt. Man hätte die Definition eines eigenen Exchanges auch umgehen können, weil ein FanoutExchange mit dem Namen *amq.fanout* immer vorhanden sein muss.

- Ein DirectExchange verschickt die Nachrichten anhand eines Keys. Beim Binding wird dann angegeben, für welche Keys die Nachricht an die Queue weitergegeben werden soll. Dadurch können beispielsweise

Aufträge an einen bestimmten Worker geschickt werden. Ein Beispiel zeigt Listing 3: Nach dem Aufbau der Queue wird ein Binding für den Key *hello*. Key zu dem DirectExchange *amq.direct* angelegt, der immer im System vorhanden sein muss. Es werden dann zwei Nachrichten verschickt: Eine hat den Key *hello*.Key und landet daher in der Queue, die andere hat keinen passenden Key und geht daher verloren.

- Ein TopicExchange nutzt ebenfalls Keys, aber er kann auch ein Muster von Keys enthalten, die relevant sind. Der Key wird dabei durch Punkte „.“ in einzelne Worte unterteilt. Ein Hashmark # in dem Muster passt dann auf ein oder mehrere Worte, ein Stern * auf genau eines. Mithilfe dieses Exchanges sind Publish/Subscribe-Szenarien möglich, bei denen Empfänger jeweils an bestimmten Nachrichten interessiert sind.
- Schließlich gibt es noch den HeadersExchange, der den Header einer Message routet. Dabei handelt es sich um verschiedene Metadaten einer Nachricht, die in die Betrachtung einbezogen werden können.

Mithilfe von Exchanges ist es also möglich, Routing-Algorithmen zu definieren. Weitere interessante Features von AMQP/RabbitMQ sind die mögliche Persistenz von Nachrichten und die Möglichkeit, einer Anfrage eine Antwort mithilfe einer Correlation ID zuzuordnen. Natürlich sind auch wiederholte Zustellungen bei einem Fehlschlag möglich und man kann Nachrichten auch quittieren. Außerdem gibt es Unterstützung für Transaktionen.

RabbitMQ und Spring

Neben dem schon vorgestellten API gibt es bei Spring AMQP natürlich auch die Möglichkeit, AMQP-Ressourcen mithilfe von Spring-XML-Dateien zu konfigurieren. Listing 4 zeigt ein Beispiel: Es wird eine *ConnectionFactory* konfiguriert, um die Verbindung zum RabbitMQ-Server herzustellen. Dann wird ein *RabbitTemplate* eingerichtet, das Zugriff auf den Server hat. Das letzte Element zeigt die Nutzung des Rabbit-XML-Namespaces, der die Konfiguration einiger RabbitMQ-Elemente vereinfacht. Damit ist es auch möglich, Queues und Exchanges einfach zu konfigurieren. In diesem Fall wird aber ein *MessageListenerContainer* konfiguriert. Er löst ein Problem beim Empfangen von Nachrichten: In den bisher gezeigten Beispielen musste man dazu die *receive()*-Methode aufrufen, sodass der aktuelle Thread blockiert wird. Günstiger wäre es, wenn ein Objekt aufgerufen wird, falls gerade eine Nachricht empfangen worden ist. Dazu dienen *MessageListenerContainer*: Sie rufen einen *MessageListener* auf, wenn eine neue Nachricht in der Queue ist. Der *MessageListener* implementiert dann die Methode *onMessage()* zur Abarbeitung der Nachricht. Der *MessageListenerContainer* enthält einen Thread Pool und andere Features, sodass er auch mit hoher Last zurechtkommt.

Aber auch diese Lösung hat noch ein Problem: Der Empfänger muss das Interface *MessageListener* implementieren und hängt daher von Spring AMQP ab.

Solche Abhängigkeiten sollten so weit wie möglich vermieden werden. Die in Listing 4 gezeigte Konfiguration ruft daher stattdessen einfach die Methode *consume()* an dem Spring Bean *consumer* auf. Diese könnte als Signatur *String consume(String message)* haben. Dann wird aus der Nachricht der Inhalt als String mithilfe eines *MessageConverters* extrahiert und als Parameter für die Methode genutzt.

Was geschieht aber mit dem Rückgabewert der Methode? Er wird als Message an die in der Nachricht mitgeschickte Reply-To-Adresse geschickt und dabei wird die Correlation ID so gesetzt, dass die Beziehung zum Request klar ist. Wenn man nun, wie in Listing 5 gezeigt, die Methode *convertSendAndReceive()* am *RabbitTemplate* aufruft, wird der Request losgeschickt. Dabei wird im Reply-To eine Queue mitgegeben, die exklusiv für den Client ist. In diese Queue wird die Antwort geschickt, konvertiert und das Ergebnis wird als Ergebnis der Methode *convertSendAndReceive()* zurückgegeben. Dadurch ist es also sehr einfach, eine Request-Response-Kommunikation aufzubauen.

Fazit

Mit AMQP steht ein Standard-Netzwerkprotokoll für Messaging zur Verfügung. Die dabei verwendeten Konzepte erlauben den Aufbau von flexiblen und komplexen Messaging-Strukturen aus einfachen Elementen (Queues und Exchanges). Spring AMQP macht die Nutzung dieses Ansatzes sehr einfach und fügt zusätzliche Flexibilität, zum Beispiel bezüglich der Kodierung der Daten in den Messages, hinzu.

Neben den hier vorgestellten Features gibt es auch eine Integration in das Spring-Integration-Projekt, und auch eine .NET-Version von Spring AMQP ist verfügbar. Wenn das Protokoll und die Skalierbarkeit näher interessieren, dem sei [6] ans Herz gelegt.



Eberhard Wolff (Twitter: @ewolff) arbeitet als Architecture & Technology Manager für die adesso AG. Seine Schwerpunkte liegen auf Cloud-Technologien, Enterprise Java u. a. mit Spring und Software Architekturen.

Links & Literatur

- [1] <https://www.amqp.org/>
- [2] Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003, ISBN 0321200683
- [3] <http://www.springsource.org/spring-amqp>
- [4] <https://www.amqp.org/confluence/display/AMQP/1-0+JMS+Mapping>
- [5] [git://github.com/pieterh/openamqp-jms.git](https://github.com/pieterh/openamqp-jms.git)
- [6] <http://blog.springsource.com/2011/04/01/routing-topologies-for-performance-and-scalability-with-rabbitmq>