RabbitMQ

platform-neutral, wire-level protocol for message-oriented middleware

Introduction

- RabbitMQ is an open-source **message-broker software** or **message-oriented middleware** that originally implemented the **Advanced Message Queuing Protocol** (AMQP).
- since been extended with a plug-in architecture to support **Streaming Text Oriented Messaging Protocol** (STOMP), **MQ Telemetry Transport** (MQTT), and other protocols.
- Written in **Erlang**, the RabbitMQ server is built on the **Open Telecom Platform** framework for **clustering** and **failover**.
- Client libraries to interface with the broker are available for all major programming languages. The source code is released under the **Mozilla Public License**

Introduction

Developer	Pivotal Software
Stable Release	3.8.19 / July 5, 2021
Repository	https://github.com/rabbitmq
Written	Erlang
Operating System	Cross-platform
Type	AMQP, message-oriented middleware
License	Mozilla Public License
Website	www.rabbitmq.com
Minimum Java Requirement	Java 1.6+

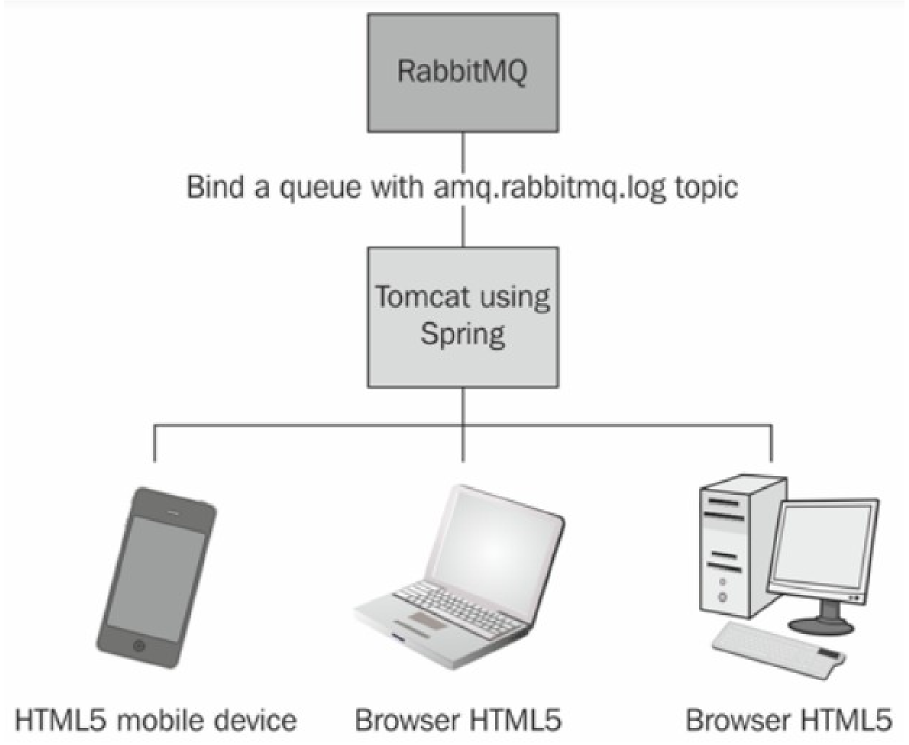
Who Developed RabbitMQ

- Originally developed by **Rabbit Technologies** Ltd. which started as a joint venture between **LShift** and **CohesiveFT** in **2007**.
- RabbitMQ was acquired in **April 2010** by SpringSource, a division of Vmware.
- The project became part of Pivotal Software in May 2013
- The broker itself, that is, the service that will actually handle the messages that are going to be sent and received by the applications

Advanced Message Queuing Protocol (AMQP)

- interoperability among the many different messaging solutions, that were developed a few years ago by many different vendors such as IBM MQ-Series, TIBCO, or Microsoft MSMQ.
- The AMQP 0-9-1 standard gives a complete specification of the protocol, particularly regarding:
  - The API interface
  - The wire protocol
- RabbitMQ is a free and complete AMQP broker implementation. It implements version 0-9-1 of the AMQP specification
- AMQP 1.0 only defines the evolution of the wire-level protocol—the format of the data being passed at the application level—for the exchange of messages between two endpoints

What Consists of



What Consists of

- The RabbitMQ exchange server
- Gateways for AMQP, HTTP, STOMP, and MQTT protocols
- AMQP client libraries for Java, Python, Ruby, .NET Framework and Erlang, C Language.
- A plug-in platform for extensibility, with a predefined collection of supported plug-ins, including:
  - A **"Shovel"** plug-in that takes care of moving or copying (replicating) messages from one broker to another.
  - A **"Federation"** plug-in that enables efficient sharing of messages between brokers (at the exchange level).
  - A **"Management"** plug-in that enables monitoring and control of brokers and clusters of brokers.
    - rabbitmq-plugins enable rabbitmq_management
  - https://www.rabbitmq.com/install-windows.html

RabbitMQ Installation

```
sudo apt-get update

sudo apt-get -y install socat logrotate init-system-helpers adduser

sudo apt-get -y install wget

sudo apt install socat

wget
https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.9.11/
rabbitmq-server_3.9.11-1_all.deb

sudo dpkg -i rabbitmq-server_3.9.11-1_all.deb

rm rabbitmq-server_3.9.11-1_all.deb

rabbitmq-plugins enable rabbitmq_management
```

RabbitMQ Confirmation

- To check the **RabbitMQ** status, use the command-line control tool ***rabbitmqctl***. It should be in the PATH in the Linux setup.
- On Windows it can be found running the **RabbitMQ command shell** by navigating to **Start Menu | All Programs | RabbitMQ Server | RabbitMQ Command Promp**t (sbin dir).
- We can run ***rabbitmqctl.bat*** from this command prompt and to check the queue status with the command ***rabbitmqclt list_queues***.
- When RabbitMQ is started for the first time, it creates some **predefined exchanges**. Command Issuing ***rabbitmqctl list_exchanges*** we can observe many existing exchanges.
- rabbitmq-plugins enable rabbitmq_management

RabbitMQ with Linux

- On Linux, RedHat, Centos, Fedora, Raspbian, and so on:

```
service rabbitmq-server restart
```

- On Linux, Ubuntu, Debian, and so on:

```
/etc/init.d/rabbitmq restart
```

- On Windows:
- `sc stop rabbitmq / sc start rabbitmq`

How to check the RabbitMQ status

- rabbitmqctl It should be in the PATH in the Linux setup. On Windows it can be found running the RabbitMQ command shell by navigating to **Start Menu** | **All Programs** | **RabbitMQ Server** | **RabbitMQ Command Prompt** (**sbin** dir). We can run **rabbitmqctl.bat** from this command prompt.
- We can check the queue status with the command **rabbitmqclt list_queues**.
- sudo service rabbitmq-server restart

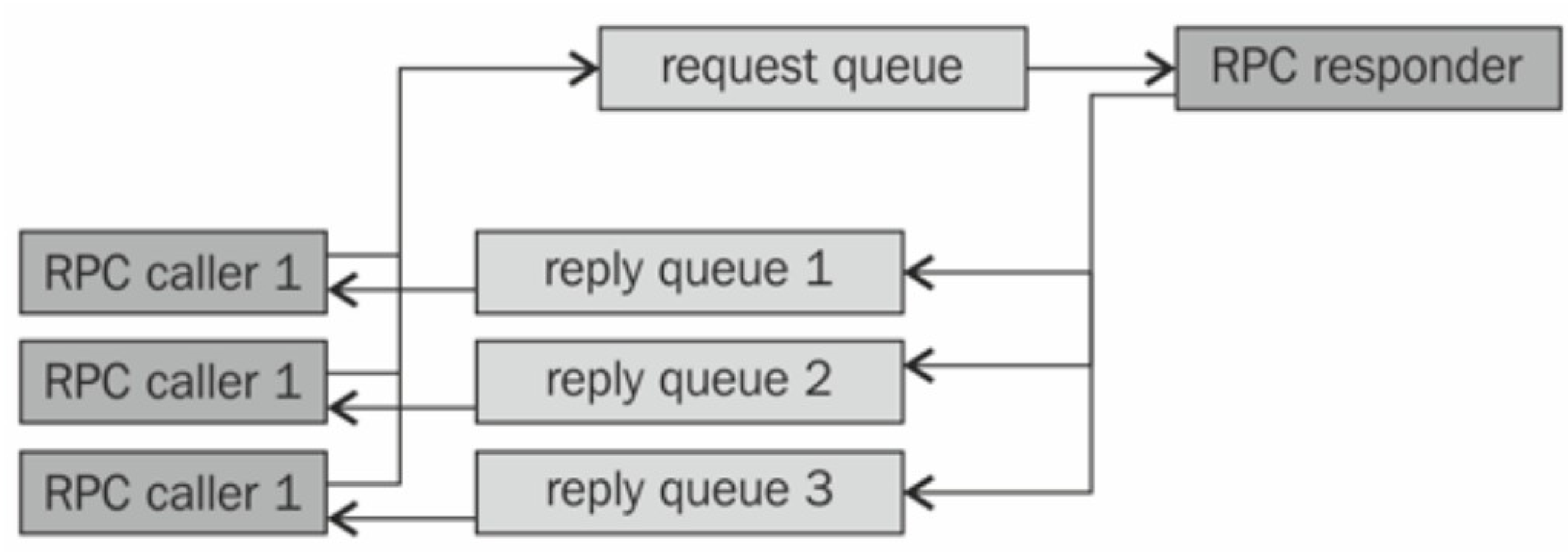
How to check the RabbitMQ status

- http://localhost:15672/
- https://www.rabbitmq.com/tutorials/tutorial-one-spring-amqp.html
-

Using RPC with messaging

- Remote Procedure Calls (RPC) are commonly used with client-server architectures. *The client is required to perform some actions to the server, and then waits for the server reply.*
- The messaging paradigm tries to enforce a totally different approach with the **fire-and-forget messaging** style, but it is possible to use properly designed AMQP queues to perform and enhance RPC
- Next Graphically, it is depicted that the request queue is associated with the responder, the reply queues with the callers.
- All the involved peers (callers and responders) are AMQP clients.

Connecting to the broker



Connecting to the broker

- RabbitMQ (as well as any other AMQP broker up to version 1.0) works over TCP as a reliable transport protocol on port 5672, that is, the IANA-assigned port.
-

implementing the RPC responder

1. Declare the request queue where the responder will be waiting for the RPC requests:

```
channel.queueDeclare(requestQueue, false, false, false,  null);
```

2. Define our specialized consumer **RpcResponderConsumer** by overriding **DefaultConsumer.handleDelivery()** as already seen in the Consuming messages recipe. On the reception of each RPC request, this consumer will:

- Perform the action required in the RPC request
- Prepare the reply message
- Set the correlation ID in the reply properties by using the following code:

```
 BasicProperties replyProperties = new  BasicProperties.Builder().correlationId(properties.
getCorrelationId()).build();
```

- Publish the answer on the reply queue:

```
 getChannel().basicPublish("", properties.getReplyTo(), replyProperties, reply.getBytes());
```

- Send the ack to the RPC request:

```
 getChannel().basicAck(envelope.getDeliveryTag(), false);
```

implementing the RPC caller

1. Declare the request queue where the responder will be waiting for the RPC requests:

```
channel.queueDeclare(requestQueue, false, false, false, null);
```

2. Create a temporary, private, autodelete reply queue:

```
String replyQueue = channel.queueDeclare().getQueue();
```

3. Define our specialized consumer RpcCallerConsumer, which will take care of

receiving and handling RPC replies. It will:

- Allow to specify what to do when it gets the replies by defining AddAction()

- Override handleDelivery():

implementing the RPC caller

```
public void handleDelivery(String consumerTag,  Envelope envelope,
AMQP.BasicProperties properties,
 byte[] body) throws java.io.IOException {
 String messageIdentifier =
 properties.getCorrelationId();
 String action = actions.get(messageIdentifier);
 actions.remove(messageIdentifier);
 String response = new String(body);
 OnReply(action, response);
 }
```

4. Start consuming reply messages invoking channel.basicConsume().

implementing the RPC caller

5. Prepare and serialize the requests (messageRequest in our example).

6. Initialize an arbitrary, unique message identifier (messageIdentifier).

7. Define what to do when the consumer gets the corresponding reply, by binding the action with the **messageIdentifier**. In our example we do it by calling our custom method **consumer.AddAction().**

8. Publish the message to requestqueue, setting its properties:

```
BasicProperties props = new
BasicProperties.Builder().correlationId(messageIdentifier).replyTo(replyQu
eue).build();
```

```
channel.basicPublish("", requestQueue,
props,messageRequest.getBytes());
```

Java Connection

1. Importing All needed library

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
```

2. Create an instance of the client ConnectionFactory:

```
ConnectionFactory factory = new ConnectionFactory();
```

3. Set the ConnectionFactory options establishing connection:

```
factory.setHost(rabbitMQhostname);
```

4. Connect to the RabbitMQ broker:

```
Connection connection = factory.newConnection();
```

5. Create a channel from the freshly created connection:

```
Channel channel = connection.createChannel();
```

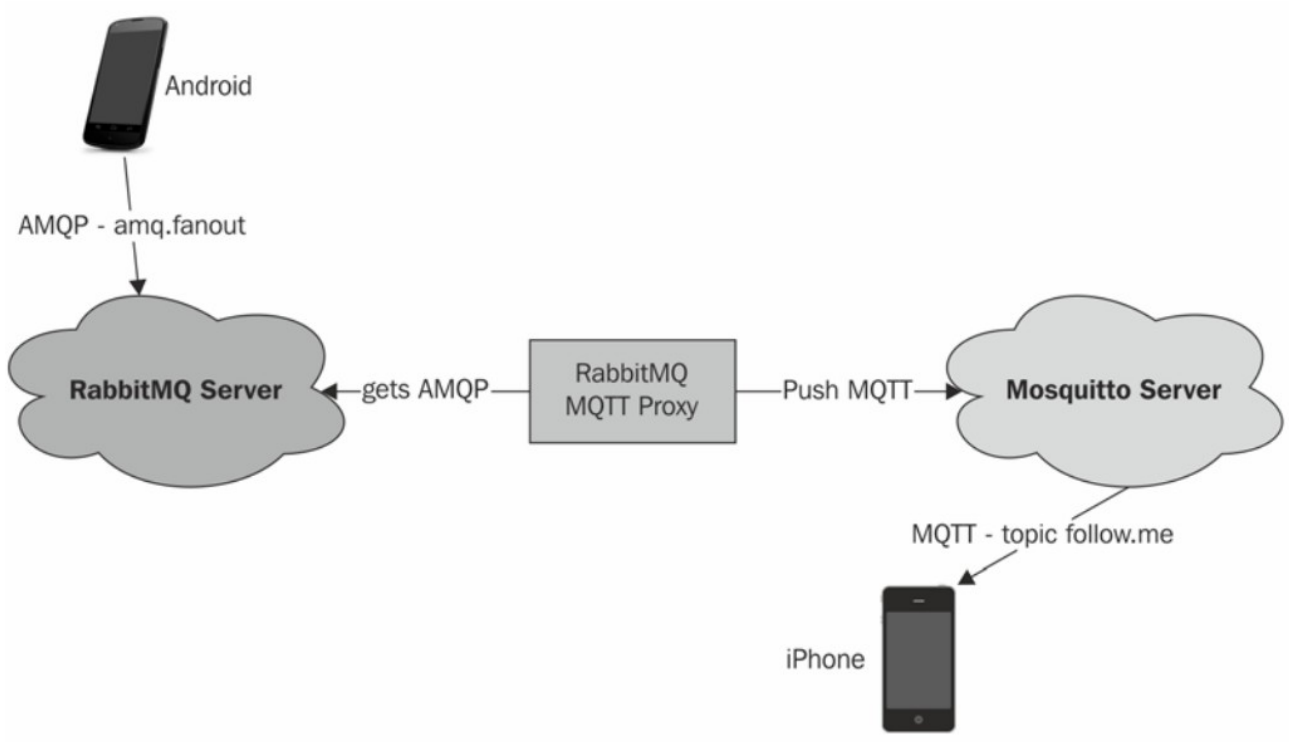
6. As soon as we are done with RabbitMQ, release the channel and the connection:

```
channel.close();
connection.close();
```

Recommendation

- If you are developing multi-threaded applications, it is highly recommended to use a different channel for each thread.
- If many threads use the same channel, they will serialize their execution in the channel method calls, leading to possible performance degradation

Recommendation



Virtual Hosts

- **Virtual hosts** are administrative containers; they allow to configure many logically independent brokers hosts within one single RabbitMQ instance, to let many different independent applications share the same RabbitMQ server.
- Each virtual host can be configured with its independent set of permissions, exchanges, and queues and will work in a logically separated environment

Setting URI

- To specify connection options by using just a connection string, also called connection URI, with the factory.setUri() method:

```
ConnectionFactory factory = new ConnectionFactory();
String uri="amqp://user:pass@hostname:port/vhost";
factory.setUri(uri);
```

Sending Message

1. Declare the queue, calling the queueDeclare() method on com.rabbitmq.client.Channel:

```
String myQueue = "myFirstQueue";
channel.queueDeclare(myQueue, true, false, false, null);
```

2. Send the very first message to the RabbitMQ broker:

```
String message = "My message to myFirstQueue";
channel.basicPublish("",myQueue, null, message.getBytes());
```

3. Send the second message with different options:

```
channel.basicPublish("",myQueue,MessageProperties.PERSISTENT_TEXT_PLAIN
,message.getBytes());
```

Sending Message

- The first step is to ensure that the destination queue exists. This task is accomplished declaring the queue (step 1) calling **queueDeclare()**.
- The method call does nothing if the queue already exists, otherwise it creates the queue itself.
- All the operations that need interactions with the broker are carried out through channels.

Arguments

- **queue:** This is just the name of the queue where we will be storing the messages.
- **durable:** This specifies whether the queue will survive server restarts. Note that it is required for a queue to be declared as durable if you want persistent messages to survive a server restart
- **exclusive:** This specifies whether the queue is restricted to only this connection.
- **autoDelete:** This specifies whether the queue will be automatically deleted by the RabbitMQ broker as soon as it is not in use.
- **arguments:** This is an optional map of queue construction arguments

Note

- The message body will never be opened by RabbitMQ.
- Messages are opaque entities for the AMQP broker, and you can use any serialization format you like.
- We often use JSON, but XML, ASN.1, standard or custom, ASCII or binary format, are all valid alternatives.
- The client applications should know how to interpret the data.

Default Properties

- **exchange** argument has been set to the empty string "", that is, the default exchange, and the **routingKey** argument to the name of the queue.
- In this case the message is directly sent to the queue specified as **routingKey**.
- The body argument is set to the byte array of our string, that is, just the message that we sent.
- The props argument is set to null as a default; these are the message properties, discussed in depth in the recipe Using message properties

Consuming Message

1. Declare the queue where we want to consume the messages from:

```
String myQueue="myFirstQueue";
channel.queueDeclare(myQueue, true, false, false, null);
```

2. Define a specialized consumer class inherited from DefaultConsumer:

```
public class ActualConsumer extends DefaultConsumer {
 public ActualConsumer(Channel channel) {
 super(channel);
 }
 @Override
 public void handleDelivery(
 String consumerTag,
 Envelope envelope,
 BasicProperties properties,
 byte[] body) throws java.io.IOException {
 String message = new String(body);
 System.out.println("Received: " + message);
 }
}
```

Consuming Message

3. Create a consumer object, which is an instance of this class, bound to our channel:

```
ActualConsumer consumer = new ActualConsumer(channel);
```

4. Start consuming messages:

```
String consumerTag = channel.basicConsume(myQueue, true,
 consumer);
```

5. Once done, stop the consumer:

```
channel.basicCancel(consumerTag);
```

Scalability notes

- What happens when there are multiple callers? It mainly works as a standard RPC client/server architecture. But what if we run many responders?
- In this case all the responders will take care of consuming messages from the request queue. Furthermore, the responders can be located on different hosts.
- We have just got load distribution for free. More on this topic is in the recipe Distributing messages to many consumers.

Broadcasting messages

- messaging application, broadcasting to a huge number of clients. For example, when updating the scoreboard in a massive multiplayer game, or when publishing news in a social network application.
- In this recipe we are discussing both the producer and consumer implementation. Since it is very typical to have consumers using different technologies and programming languages, we are using Java, Python, and Ruby to show interoperability with AMQP.

Managing RabbitMQ from a browser

- we're showing you how to admin RabbitMQ from an HTTP API using a Management Plugin.The plugin provides real-time charts to monitor the flow of your messages. Furthermore, it provides HTTP APIs to analyze RabbitMQ.
- This is required by external monitoring systems such as Ganglia (http://ganglia.sourceforge.net/), Puppet (http://puppetlabs.com), and others in order to perform their activities Issue the following command:

```
rabbitmq-plugins enable rabbitmq_management
```

- 2. Restart RabbitMQ.
- 3. The plugin enables a web server that is accessible via the URL http://localhost:15672/. Replace localhost with your RabbitMQ hostname/IP to access from another machine.

Publishing messages from Android in the background

- The Android SDK tools (http://developer.android.com/sdk/index.html)
- An Android device with Android API at least at level 11 (Honeycomb and Android 3.0.x or higher) Google Play installed on the Android device From the Android SDK Manager, install Google Play services
- 2. Create a new Android application. Following the wizard, the minimum required SDK must be set to API11. Create a Blank Activity.
- 3. Carefully follow the instructions provided by Google to set up the Google Play services (http://developer.android.com/google/play-services/setup.html).
- 4. Copy the RabbitMQ library files (c**ommons-cli-1.1.jar, commons-io-1.2.jar, and rabbitmq-client.jar**) in the libs directory of the project

application.properties

- spring.rabbitmq.addresses=amqp://admin:
  secret@localhost
- spring.rabbitmq.host=localhost
- spring.rabbitmq.port=5672
- spring.rabbitmq.username=admin
- spring.rabbitmq.password=secret

Java Code to send Message

```
public class Send {
  private final static String QUEUE_NAME = "hello";
public static void main(String[] args) throws IOException, TimeoutException {
  ConnectionFactory factory = new ConnectionFactory();
  factory.setHost("localhost");
  try (Connection connection = factory.newConnection();
           Channel channel = connection.createChannel()) {
      channel.queueDeclare(QUEUE_NAME, false, false, false, null);
      String message = "Hello World!";
      channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
      System.out.println(" [x] Sent '" + message + "'");
}}}
```

Java Code to send Message

```
@Component
public class MyBean {
    private final AmqpAdmin amqpAdmin;
    private final AmqpTemplate amqpTemplate;
    public MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {
        this.amqpAdmin = amqpAdmin;
        this.amqpTemplate = amqpTemplate;
    }
    public void someMethod() {
        this.amqpAdmin.getQueueInfo("someQueue");
    }
    public void someOtherMethod() {
        this.amqpTemplate.convertAndSend("hello");
    }
}
```

Java Code to Receive Message

```
public class Recv {
private final static String QUEUE_NAME = "hello";
public static void main(String[] args) throws IOException, TimeoutException {
ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println(" [x] Received '" + message + "'");
    };
    channel.basicConsume(QUEUE_NAME, true, deliverCallback, consumerTag -> { });
}}
```

Java Code to Receive Message

```
spring.rabbitmq.template.retry.enabled=true

spring.rabbitmq.template.retry.initial-interval=2s

@Component

public class MyBean {

    @RabbitListener(queues = "someQueue")

    public void processMessage(String content) {

        // ...

    }

}
```

Java commands

- javac -cp amqp-client-5.7.1.jar Send.java Recv.java
- java -cp .:amqp-client-5.7.1.jar:slf4j-api-1.7.26.jar:slf4j-simple-1.7.26.jar Recv
- java -cp .:amqp-client-5.7.1.jar:slf4j-api-1.7.26.jar:slf4j-simple-1.7.26.jar Send
- export CP=.:amqp-client-5.7.1.jar:slf4j-api-1.7.26.jar:slf4j-simple-1.7.26.jar java -cp $CP Send
- set CP=.;amqp-client-5.7.1.jar;slf4j-api-1.7.26.jar;slf4j-simple-1.7.26.jar java -cp %CP% Send

RabbitMQ Configuration

```
@Configuration(proxyBeanMethods = false)
public class MyRabbitConfiguration {
    @Bean
    public SimpleRabbitListenerContainerFactory myFactory(SimpleRabbitListenerContainerFactoryConfigurer configurer) {
        SimpleRabbitListenerContainerFactory factory = new SimpleRabbitListenerContainerFactory();
        ConnectionFactory connectionFactory = getCustomConnectionFactory();
        configurer.configure(factory, connectionFactory);
        factory.setMessageConverter(new MyMessageConverter());
        return factory;
    }
    private ConnectionFactory getCustomConnectionFactory() {
        return ...
    }
}
```

RabbitMQ Commands

- rabbitmqctl.bat list_queues
- sudo rabbitmqctl list_queues
- rabbitmqctl list_exchanges
- rabbitmqctl list_bindings

Java producer and a Python consumer

1. Java: In addition to the standard import described in the recipe Connecting to the broker, we have to import:

```
Import com.rabbitmq.tools.json.JSONWriter;
```

2. Java: We create a queue that is not persistent:

```
String myQueue="myJSONBodyQueue_4";
channel.queueDeclare(MyQueue, false, false, false, null);
```

Java producer and a Python consumer

3. Java: We create a list for the Book class and fill it with example data:

```
List<Book>newBooks = new ArrayList<Book>();
 for (inti = 1; i< 11; i++) {
 Book book = new Book();
 book.setBookID(i);
 book.setBookDescription("History VOL: " + i );
 book.setAuthor("John Doe");
 newBooks.add(book);
}
```

Java producer and a Python consumer

4. Java: We are ready to serialize the newBooks instance with JSONwriter:

```
JSONWriter rabbitmqJson = new JSONWriter();
String jsonmessage = rabbitmqJson.write(newBooks);
```

5. Java: We can finally send our jsonmessage:

```
channel.basicPublish("",MyQueue,null, jsonmessage.getBytes());
```

6. Python: To use the Pika library we must add the follow import:

```
import pika;
import json;
```

Java producer and a Python consumer

7. Python: In order to create a connection to RabbitMQ, use the following code:

```
connection = pika.BlockingConnection(pika.ConnectionParameters(rabbitmq_ host));
```

8. Python: Let's declare a queue, bind as a consumer, and then register a callback:

```
channel = connection.channel()
my_queue = "myJSONBodyQueue_4"
channel.queue_declare(queue=my_queue)
channel.basic_consume(consumer_callback, queue=my_queue,
 no_ack=True)
channel.start_consuming()
```

Broadcasting messages

we are preparing four different codes:

- The Java publisher
- The Java consumer
- The Python consumer
- The Ruby consumer

A Java publisher

1. Declare a fanout exchange:

```
channel.exchangeDeclare(myExchange, "fanout");
```

2. Send one message to the exchange:

```
channel.basicPublish(myExchange, "", null,
 jsonmessage.getBytes());
```

A Java consumer

1. Declare the same fanout exchange declared by the producer:

`channel.exchangeDeclare(myExchange, "fanout");`

2. Autocreate a new temporary queue:

`String queueName = channel.queueDeclare().getQueue();`

3. Bind the queue to the exchange:

`channel.queueBind(queueName, myExchange, "");`

4. Define a custom, non-blocking consumer, as already seen in the Consuming messages recipe.

5. Consume messages invoking `channel.basicConsume()`

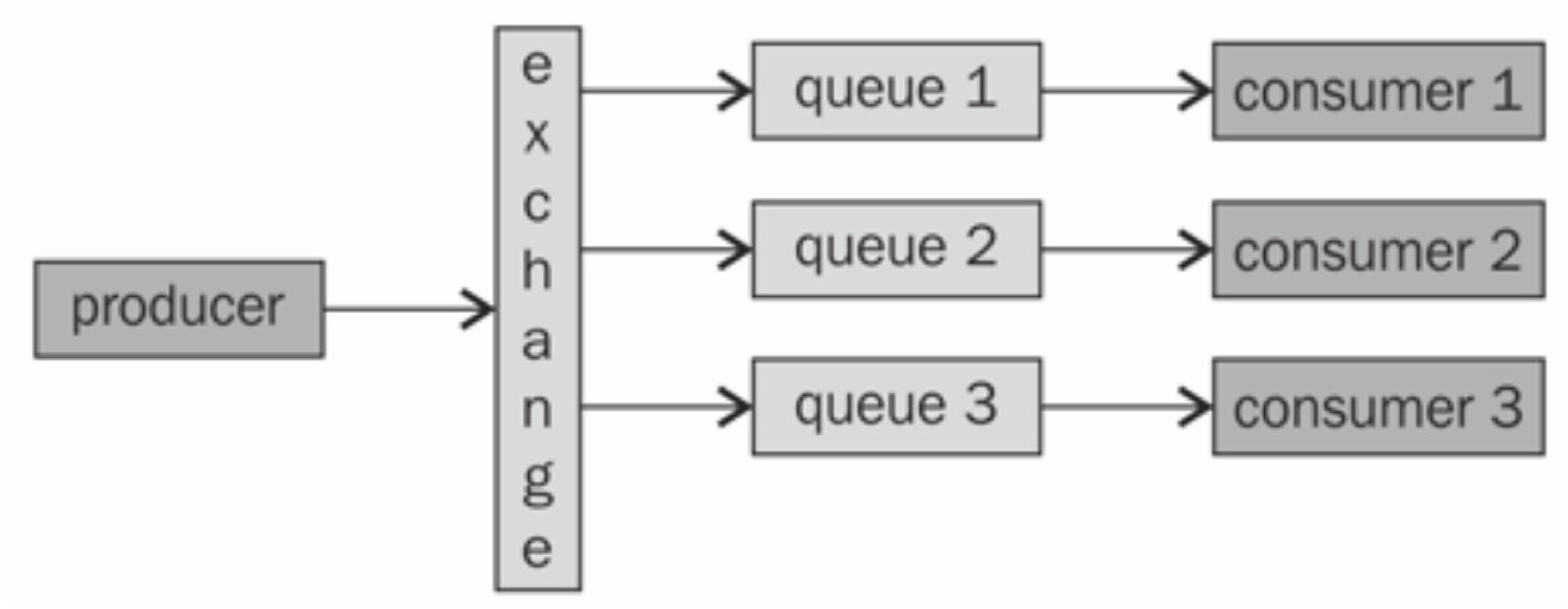
Lets Start the Action

1. Start one instance of the Java producer; messages start getting published immediately.

2. Start one or more instances of the Java/Python/Ruby consumer; the consumers receive only the messages sent while they are running.

3. Stop one of the consumers while the producer is running, and then restart it; we can see that the consumer has lost the messages sent while it was down.

Lets Start the Action



implementing both the producer and the consumer (Producer)

1. Declare a direct exchange:

```
channel.exchangeDeclare(exchangeName, "direct", false,
 false, null);
```

2. Send some messages to the exchange, using arbitrary routingKey values:

```
channel.basicPublish(exchangeName, routingKey, null,
 jsonBook.getBytes());
```

implementing both the producer and the consumer (Consumer)

1. Declare the same exchange, identical to what was done in step 1.

2. Create a temporary queue:

```
String myQueue = channel.queueDeclare().getQueue();
```

3. Bind the queue to the exchange using the **bindingKey**. Perform this operation as many times as needed, in case you want to use more than one binding key:

```
channel.queueBind(myQueue,exchangeName,bindingKey);
```

4. After having created a suitable consumer object, start consuming messages as already seen in the Consuming messages recipe.

message routing using topic exchanges(Producer)

1. Declare a topic exchange:

```
channel.exchangeDeclare(exchangeName, "topic", false, 
 false, null);
```

2. Send some messages to the exchange, using arbitrary routingKey values:

```
channel.basicPublish(exchangeName, routingKey, null, 
 jsonBook.getBytes());
```

message routing using topic exchanges(Consumer)

1. Declare the same exchange, identical to what was done in step 1.

2. Create a temporary queue:

```
String myQueue = channel.queueDeclare().getQueue();
```

3. Bind the queue to the exchange using the binding key, which in this case can

contain wildcards:

```
channel.queueBind(myQueue,exchangeName,bindingKey);
```

4. After having created a suitable consumer object, start consuming messages as already seen in the Consuming messages recipe.

Guaranteeing message processing

- The explicit acknowledgment, the so-called **ack**, while consuming messages.
- A message is stored in a queue until one consumer gets the message and sends the **ack** back to the broker.
- The **ack** can be either implicit or explicit.
- To demonstration, run the publisher from the Producing messages recipe and the consumer who gets the message.

Guaranteeing message processing

1. Declare a queue:

```
channel.queueDeclare(myQueue, true, false, false,null);
```

2. Bind the consumer to the queue, specifying false for the *autoAck* parameter of **basicConsume()**:

```
ActualConsumer consumer = new ActualConsumer(channel);
boolean autoAck = false; // n.b.
channel.basicConsume(MyQueue, autoAck, consumer);
```

3. Consume a message and send the ack:

```
public void handleDelivery(String consumerTag, Envelope envelope, BasicPropertiesproperties,byte[]
body) throws  java.io.IOException {
            String message = new String(body);
            this.getChannel().basicAck(envelope.getDeliveryTag(),false);
```

Distributing messages to many consumers

To create a dynamic ***load balancer****, and how to distribute messages to many consumers. We are going to create a file downloader. two or more RabbitMQ clients properly balance consuming messages*

1. Declare a named queue, and specify the basicQos as follows:

```
channel.queueDeclare(myQueue, false, false, false,null);
channel.basicQos(1);
```

2. Bind a consumer with explicit ack:

```
channel.basicConsume(myQueue, false, consumer);
```

3. Send one or more messages using channel.basicPublish().

4. Execute two or more consumers.

Down-loader

The publisher sends a message with the URL to download:

```
String messageUrlToDownload=  "http://www.rabbitmq.com/releases/rabbitmq-dotnet-client/v3.0.2/rabbitmq-dotnet-client-3.0.2-user-guide.pdf";
```

```
channel.basicPublish("",MyQueue,null,messageUrlToDownload.getBytes ());
```

The consumer gets the message and downloads the referenced URL:

```
System.out.println("Url to download:" + messageURL);
```

```
downloadUrl(messageURL);
```

Once the download is terminated, the consumer sends the ack back to the broker and is ready to download the next one:

```
getChannel().basicAck(envelope.getDeliveryTag(),false);
```

```
System.out.println("Ack sent!");
```

```
System.out.println("Wait for the next download...");
```

Using message properties

How an AMQP message is divided, and how to use message properties.

In order to access the message properties you need to perform the following steps:

1. Declare a queue:

```
channel.queueDeclare(MyQueue, false, false,
false,null);
```

Using message properties

2. Create a BasicProperties class:

```
Map<String,Object>headerMap = new HashMap<String, Object>();
headerMap.put("key1", "value1");
headerMap.put("key2", new Integer(50) );
headerMap.put("key3", new Boolean(false));
headerMap.put("key4", "value4");
BasicProperties messageProperties = new  BasicProperties.Builder().timestamp(new Date()).contentType("text/plain").userId("guest").appId("app id: 20").deliveryMode(1).priority(1).headers(headerMap).clusterId("cluster id: 1").build();
```

Using message properties

3. Publish a message with basic properties:

```
channel.basicPublish("",myQueue,messageProperties,message.getBytes())
```

4. Consume a message and print the properties:

```
System.out.println("Property:" +
properties.toString());
```

Using message properties

The AMQP message (also called content) is divided into two parts:

1) **Content header**
2) **Content body (as we have already seen in previous examples)**

In step 2 we create a content header using BasicProperties:

```
Map<String,Object>headerMap = new HashMap<String, Object>();
```

```
BasicProperties messageProperties = new BasicProperties.Builder().
timestamp(new Date()). userId("guest"). DeliveryMode(1). priority(1).
headers(headerMap).build();
```

Using message properties

- **timestamp:** This is the message time stamp.
- **userId:** This is the broker with whom the user sends the message (by default, it is "guest").
- **deliveryMode:** If set to 1 the message is non-persistent, if it is 2 the message is persistent.
- **priority:** This defines the message priority, which can be 0 to 9.
- **headers:** A HashMap<String, Object> header, you are free to use it to enter your custom fields.

Messaging with transactions

- In the Producing messages recipe we have seen how to use a persistent message, but if the broker can't write the message to the disk, you can lose the message.
- With the AQMP transactions you can be sure that the message won't be lost

Messaging with transactions

1. Create a persistent queue

```
channel.queueDeclare(myQueue, true, false, false, null);
```

2. Set the channel to the transactional mode using:

```
channel.txSelect();
```

3. Send the message to the queue and then commit the operation:

```
channel.basicPublish("", myQueue, MessageProperties.PERSISTENT_TEXT_PLAIN,
message.getBytes());
channel.txCommit();
```

After creating a persistent queue (step 1), we have set the channel in the transaction mode using the method **txSelect()** (step 2). Using **txCommit()** the message is stored in the queue and written to the disk; the message will then be delivered to the consumer(s).

Handling unroutable messages

- An unroutable message is a message without a destination. For example, a message sent to an exchange without any bound queue.
- Unroutable messages are not similar to dead letter messages; the first are messages sent to an exchange without any suitable queue destination.
- The latter, on the other hand, reach a queue but are rejected because of an explicit consumer decision, expired TTL, or exceeded queue length limit.

Handling unroutable messages

1. First of all we need to implement the class ReturnListener and its interface:

```
public class HandlingReturnListener implements ReturnListener {
@Override
 public void handleReturn…
```

2. Add the HandlingReturnListener class to the channel ReturnListener:

```
channel.addReturnListener(new HandlingReturnListener());
```

3. Then create an exchange:

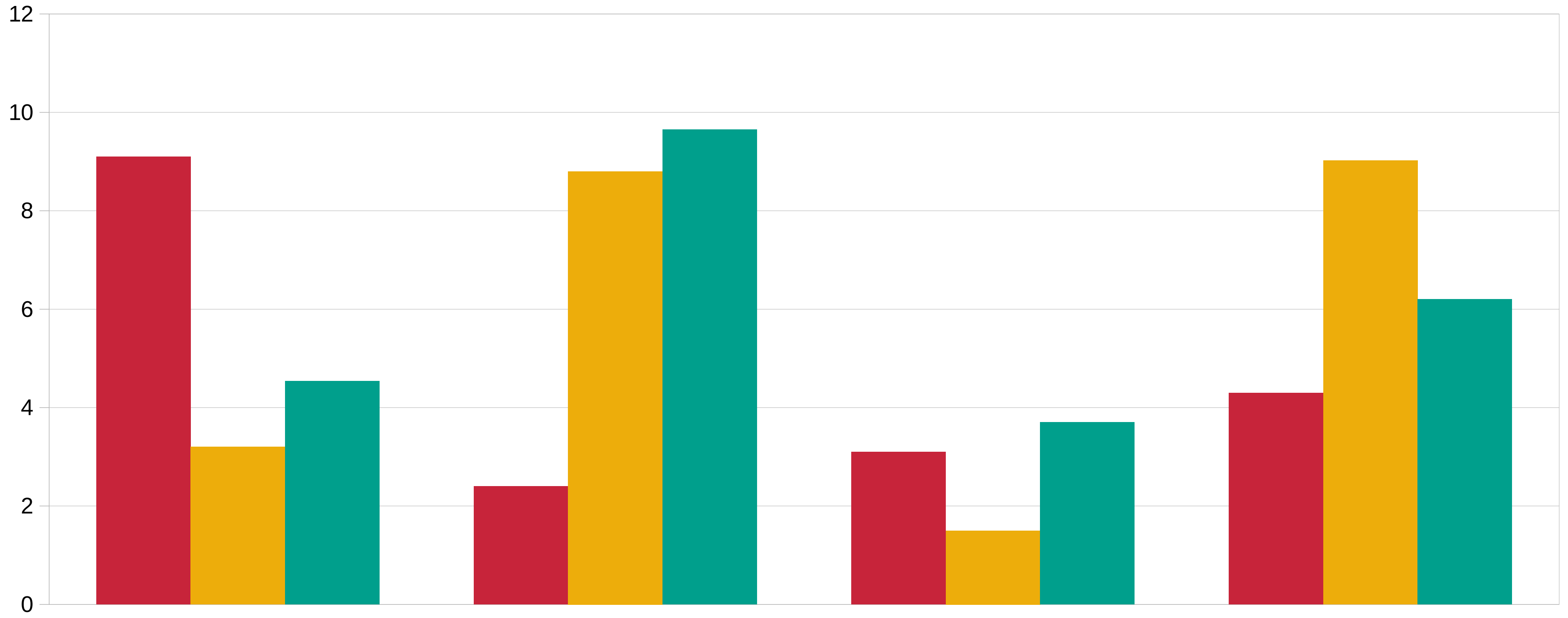
```
channel.exchangeDeclare(myExchange, "direct", false, false, null);
```

4. And finally publish a mandatory message to the exchange:

```
boolean isMandatory = true;
channel.basicPublish(myExchange, "",isMandatory, null, message.getBytes());
```

Handling unroutable messages

- When we execute the publisher, the messages sent to myExchange won't reach any destination since it has no bound queues.
- These messages aren't, they are redirected to an internal queue. The HandlingReturnListener class will handle such messages using **handleReturn()**.
- The **ReturnListener** class is bound to a publisher channel, and it will trap only its own unroutable messages.
- You can also find a consumer in the source code example. Try also to execute the publisher and the consumer together, and then stop the consumer.


12
10
8
6
4
2
0