THE

OPTIMAL RABBITMQ GUIDE

FROM BEGINNER TO ADVANCED

BY LOVISA JOHANSSON

OVER 12,000+
READERS & COUNTING

The Optimal RabbitMQ Guide

From Beginner to Advanced

84codes AB

1st Edition

Acknowledgments

I want to say a big thank you to everyone who has helped me, from the earliest draft of this book, up to this first edition. A special thanks go out to my lovely colleagues at 84codes and to my computer nerd (and IRL) friends; Emil, Filip, Lisa, and Thomas.
Finally: A big thank you, to all CloudAMQP users for your feedback and continued support. It has been a great motivator to see our customers' projects succeed!

I'd love to hear from you

I encourage you to e-mail me any comments that you might have about the book. Tell us what you think should or shouldn't be included in the next edition. If you have an application which is using RabbitMQ, and a user story that you would like to share; please send me an e-mail!

Book version: 1.0

ISBN: 978-91-639-9585-9

Author: Lovisa Johansson

Email: lovisa@cloudamqp.com

Published: 2018-10-25

Graphics: Elin Vinka, Daniel Marklund

This book is dedicated to all the frequent and future users of RabbitMQ, and to the Swedish mentality that made us queue lovers by heart.

Table of Contents

Introduction

Times are changing! Reliability and scalability are more important than ever. Because of that companies are rethinking their architecture. Monoliths are evolving into microservices and servers are moving into the cloud. Message Queues and RabbitMQ, one of the most widely deployed open source message brokers, has come to play a significant role in the growing world of microservices.

This book is divided into three parts:

The first part focuses on giving a gentle introduction to microservices and RabbitMQ, and how to use RabbitMQ on the CloudAMQP online platform. You will learn the most important RabbitMQ concepts and how RabbitMQ allows users to create products that match all of the demands made by the industry of both today and tomorrow.

The second part of the book is for the more advanced users. You will, after this part, be able to take full advantage of RabbitMQ. We talk about Best Practice recommendations for High Performance and High Availability and common RabbitMQ errors and mistakes. It's also a deep dive into some RabbitMQ concepts and features.

The third part gives some real-world user stories from own experiences and a CloudAMQP customer's point of view.

We hope that this book will help to take you even further on your message queuing journey.

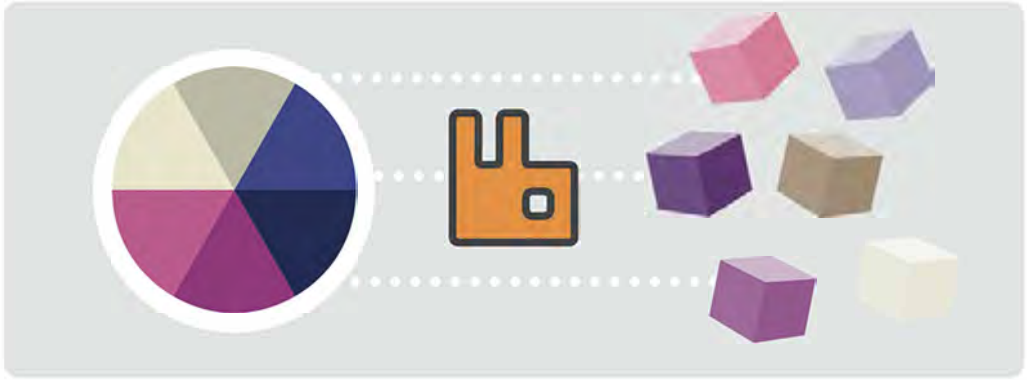
Part 1: Introduction to RabbitMQ

Welcome to the wonderful world of message queuing!

In many modern cloud architectures, applications are decoupled into small, independent applications - that together build up a microservice architecture; A collection of loosely coupled services, that communicate via a message queue. Through a microservice architecture, different parts of the system are easier to develop, deploy and maintain.

One of the advantages of a message queue is that a queue makes your data temporarily persistent, and reduce the errors that might happen when different parts of your system are offline. If one part of the system is unreachable, the other part can continue to interact with the queue. This part of the book gives you a kick-start to microservices, message queuing and RabbitMQ.

Microservices and RabbitMQ



Managing a complex, unified enterprise application can be a lot more difficult than it seems. Before making a small change, you might need to spend hours of brainstorming and analyze possible impacts your change could have on the overall system. Also, new developers have to spend days getting familiar with the system before they are considered ready to write a non-breaking line of code. The situation could be less complicated, through the use of microservices. This chapter talks about the benefits of a microservice architecture.

In a microservice architecture (also called modular architecture) different modules are de-coupled from each other. These modules often offer various functionalities, forming a complete software application, via properly articulated operations and interfaces. Unlike a monolithic application, where all chunks of functionality are present within a single module, a microservice application divides different functionalities across different modules, to enhance maintainability and ease-of-use. More and more organizations are going down the microservices road because it allows them to have a more agile approach to software development and maintenance. Microservices also make it easy for businesses to scale and deliver updated versions of software weekly, instead of yearly.

Benefits of a microservice architecture

- **Development and maintenance made easier**
  Imagine that you have to build a huge, bulky billing application which will involve authentication, authorization, charging and reporting of transactions. If you divide your application across multiple services (one for each functionality), you will be able to separate the responsibilities and also give developers the freedom to write code for a specific service in any chosen language; something that a monolithic application can't provide. Additionally, it will also be easier to maintain written code and make changes to the system. For example, if you are asked to update your authentication scheme, you will only have to add code to the authentication module and test it, without having to worry about disrupting any other functionalities.

- **Fault isolation**
  Another obvious advantage offered by a microservice architecture is its ability to isolate the fault to a single module. For example, if your reporting service is down, your authentication and

billing services will still be up, ensuring that your customers can perform important transactions even when they aren't able to view reports.

- **Enhanced levels of speed and productivity**
  Microservices is all about fragmenting applications into manageable modules and functionalities that are easy to maintain. Different developers can work on different modules at the same time. In addition to the development cycle, the testing phase is also sped up by the use of microservices, as you can test each microservice on its own to determine the readiness of the overall system.

- **Improved scalability**
  Microservices also allow you to scale effortlessly and at will. If you want to add new components to just one service, you can do that without changing any of the other services. You can also deploy the more resource-intensive services across multiple servers.

- **Easy to understand**
  Another advantage offered by a microservice application is the ease of understandability. Since each module represents a single functionality, getting to know the relevant details becomes easier and faster. For example, if you have hired a consultant for your charging services, they don't have to understand the whole system in order to offer you insight; they only have to familiarize themselves with one module to do so.

The role of a message queue in a microservice architecture

Typically, in a microservice architecture, there are cross-dependencies, which entail that no single service can perform its functionalities without getting help from other services. This is where it's crucial for your system to have a mechanism in place which allows services to keep in touch with each other without getting blocked by responses. Message queuing fulfills this purpose by providing a means for services to push messages to a queue asynchronously and ensure that they get delivered to the correct destination. To implement message queuing, you need a message broker; think of it as a mailman, who takes mail from a sender and delivers it to the correct destination.

What is RabbitMQ?

Message queuing is an important facet of any microservice architecture, and it's a way of exchanging data between processes, applications and servers. RabbitMQ is one of the most widely used message brokers, with over 35,000 production deployments worldwide, across startups and big enterprises alike. This chapter gives a brief understanding of messaging and defines essential RabbitMQ concepts. The chapter also explains the steps to go through when setting up a connection and how to publish or consume messages from a queue.

RabbitMQ is a message-queuing software called a message broker or queue manager. Simply put; it's a software where queues can be defined, and applications may connect to the queue and transfer a message onto it. Message queues enable asynchronous communication, which means that other applications (endpoints) that are producing and consuming messages interact with the queue, instead of communicating directly with each other.

A message can include any information. It could, for example, contain information about a process/job that should start on another application (that could be on another server), or it might as well just be a simple text message.

Figure 1 - Messages sent from a sender to a receiver.

The message broker stores the messages until a receiving application connects and takes a message off the queue. The receiving application then appropriately processes the message. A producer can add messages to a queue without having to wait for them to be processed.

RabbitMQ Example

A message broker can act as a middleman for various services. A broker can be used to reduce loads and delivery times by web application servers since tasks that would generally take a lot of time to process can be delegated to a third party whose only job is to perform them.

In this chapter, we follow a scenario where a web application allows users to upload information to a website. The site handles this information and generates a PDF and email it back to the user. In this example case, processing the data, creating the PDF, and sending the email will take several seconds.

When the user has entered user information into the web interface, the web application puts a "PDF processing"-job into a message (including information about the job), and the message is placed onto a queue defined in RabbitMQ. The underlying architecture of a message queue is simple: there are client applications called producers that create messages and deliver them to the broker (the message queue).

Other applications, called consumers, connects to the queue and subscribes to messages from the broker, and processes those messages. The software interacting with the broker can be a producer, a consumer, or both a consumer and a producer of messages. Messages placed in the queue are stored until the consumer retrieves them.

If the PDF processing application crashes, or if many PDF requests are coming at the same time, messages would continue to pile up in the queue until the consumer starts again. It would then process all the messages, one by one.

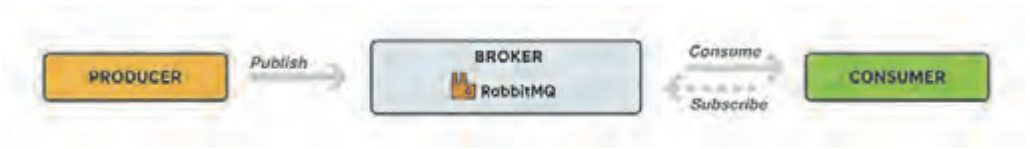


Figure 2 - A sketch of the RabbitMQ workflow.

Why and when should you use RabbitMQ?

Message queuing allows web servers to respond to requests in their own time, instead of being forced to perform resource-heavy procedures on the spot. Message queuing is also useful when you want to distribute a message to multiple recipients for consumption or when you need to balance the load between workers.

The consumer can remove a message from the queue and start the processing of the PDF at the same time as the producer is pushing new messages to the queue. The consumer can be on an entirely different server than the publisher, or they can be located on the same server, it makes no difference. Requests can be created in one programming language and handled in another programming language - the two applications only communicate through the messages they are sending to each other. Because of that we say that the two services have low coupling between the sender and the receiver.

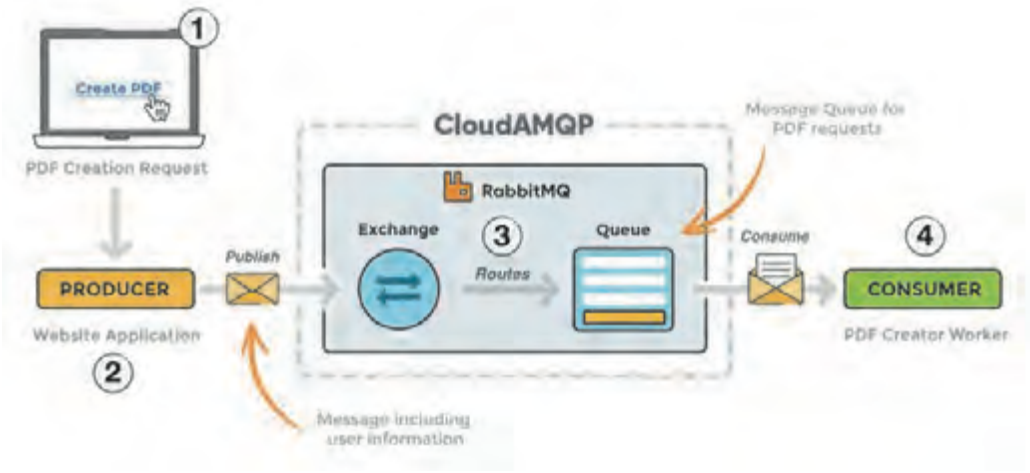


Figure 3 - Application architecture example with RabbitMQ.

Example

Figure 3 shows an application architecture example with RabbitMQ.

1. The user sends a PDF create request to the web application.

2. The web application (the producer) sends a message to RabbitMQ, including data from the request, such as name and email.

3. An exchange accepts the messages from a producer application and routes them to correct message queues.

4. The PDF processing worker (the consumer) receives the job and starts the processing of the PDF.

Exchanges

Messages are not published directly to a queue. Instead, the producer sends a message to an exchange. An exchange is responsible for the routing of messages to different queues. The job of an exchange is to accept messages from the producer applications and route them to the correct message queues. It does this with the help of bindings and routing keys. A binding is a link between a queue and an exchange.

The message flow in RabbitMQ

Figure 4 illustrates the message flow in RabbitMQ.

1. The producer publishes a message to an exchange. When you create the exchange, you have to specify the type of it. The different types of exchanges are explained in detail later on.

2. The exchange receives the message and is now responsible for the routing of the message. The exchange looks at different message attributes and keys depending on the exchange type.

3. In this case, we see two bindings to two different queues from the exchange. The exchange routes the message to the correct queue, depending on its attributes.

4. The messages stay in the queue until a consumer handles them.

5. The consumer handles the message, thus removing it from the queue.

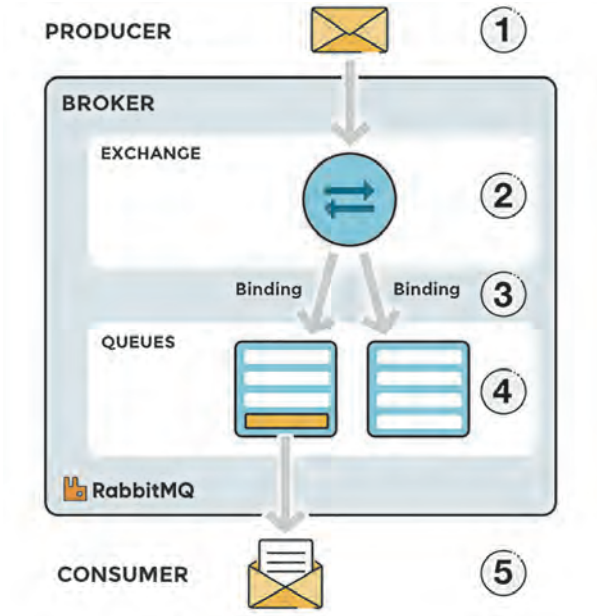


Figure 4 - Illustration of the message flow in RabbitMQ, with five steps highlighted.

Types of exchanges

Only direct exchanges are used within our sample code in the upcoming chapters. More in-depth understanding of the different exchange types, binding keys, routing keys and how/when they should be used can be found in the chapter: Exchanges, routing keys and bindings.

- **Direct**: A direct exchange delivers messages to queues based on a message routing key. In a direct exchange, the message is routed to the queue whose binding key exactly matches the routing key of the message. If the queue is bound to the exchange with the binding key pdfprocess, a message published to the exchange with a routing key pdfprocess is routed to that queue.

- **Topic**: The topic exchange performs a wildcard match between the routing key and the routing pattern specified in the binding.

- **Fanout**: A fanout exchange routes messages to all of the queues that are bound to it.

- **Headers**: Headers exchanges use the message header attributes to do their routing.

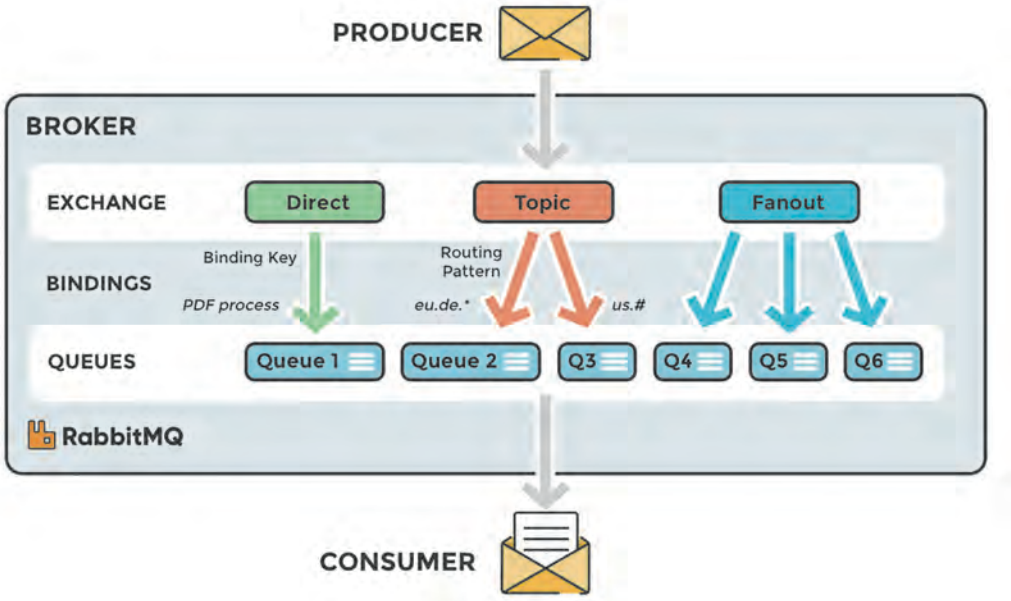


Figure 5 - Three different exchanges: direct, topic, and fanout.

RabbitMQ and server concepts

Here are some important concepts that are helpful to know before we dig deeper into RabbitMQ. The default virtual host, the default user, and the default permissions are used in the examples that follow.

- **Producer:** Application that sends the messages.
- **Consumer:** Application that receives the messages.
- **Queue:** Buffer that stores messages.
- **Message:** Data that is sent from the producer to a consumer through RabbitMQ.
- **Connection:** A connection is a TCP connection between your application and the RabbitMQ broker.
- **Channel:** A channel is a virtual connection inside a connection. When you are publishing or consuming messages or subscribing to a queue, it's all done over a channel.
- **Exchange:** Receives messages from producers and pushes them to queues depending on rules defined by the exchange type. A queue needs to be bound to at least one exchange to be able to receive messages.
- **Binding:** A binding is a link between a queue and an exchange.

- **Routing key:** The routing key is a key that the exchange looks at to decide how to route the message to queues. You can think of the routing key as the destination address of a message.
- **AMQP:** The Advanced Message Queuing Protocol is the primary protocol used by RabbitMQ for messaging.
- **Users:** It's possible to connect to RabbitMQ with a given username and password. Every user can be assigned permissions such as rights to read, write and configure privileges within the instance. Users can also be assigned permissions to specific virtual hosts.
- **Vhost, virtual host:** A virtual host provides a way to segregate applications that are using the same RabbitMQ instance. Different users can have different access privileges to different vhosts and queues, and exchanges can be created so that they only exist in one vhost.
- **Acknowledgments and Confirms:** Acknowledgments and confirms indicate that messages have been received or acted upon. Acknowledgments can be used in both directions; A consumer can indicate to the server that it has received/processed a message, and the server could report the same thing to the producer.

Setting up a RabbitMQ instance

To be able to follow this guide you need to set up a CloudAMQP instance or set up RabbitMQ on your local machine. CloudAMQP is a hosted RabbitMQ solution (RabbitMQ as a Service), meaning that all you need to do is sign up for an account and create an instance. You do not need to set up and install RabbitMQ or care about cluster handling. CloudAMQP will do all of that for you. It is also available for free with the plan **Little Lemur**. Go to the plan page (www.cloudamqp.com/plans.html) and sign up for a plan of your choice. Click on "details" for your cloud-hosted RabbitMQ instance to find your username, password and connection URL (*figure 7*).

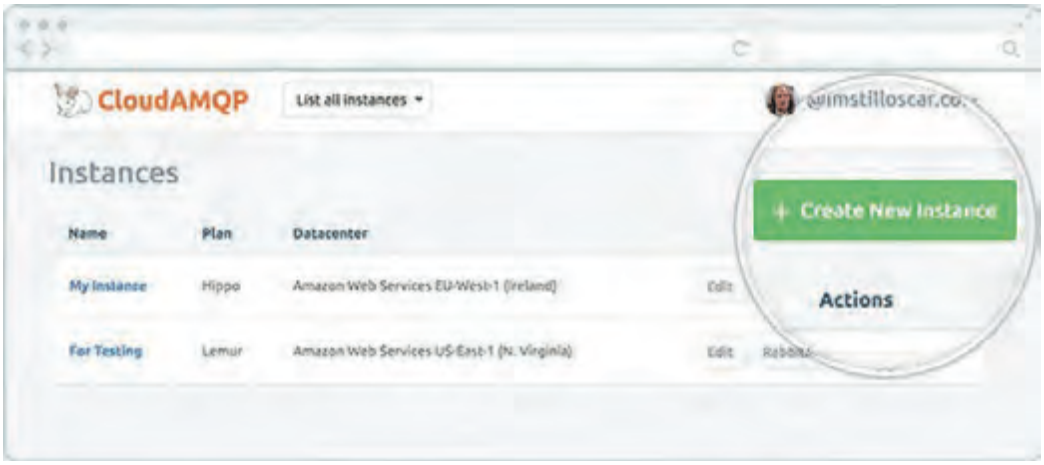


Figure 6 - Instances in the CloudAMQP web interface.

Getting started with RabbitMQ

It's possible to send messages across languages, platforms, and operating systems once you have the RabbitMQ instance up and running. You can now start by opening the management interface to get an overview of your RabbitMQ server.

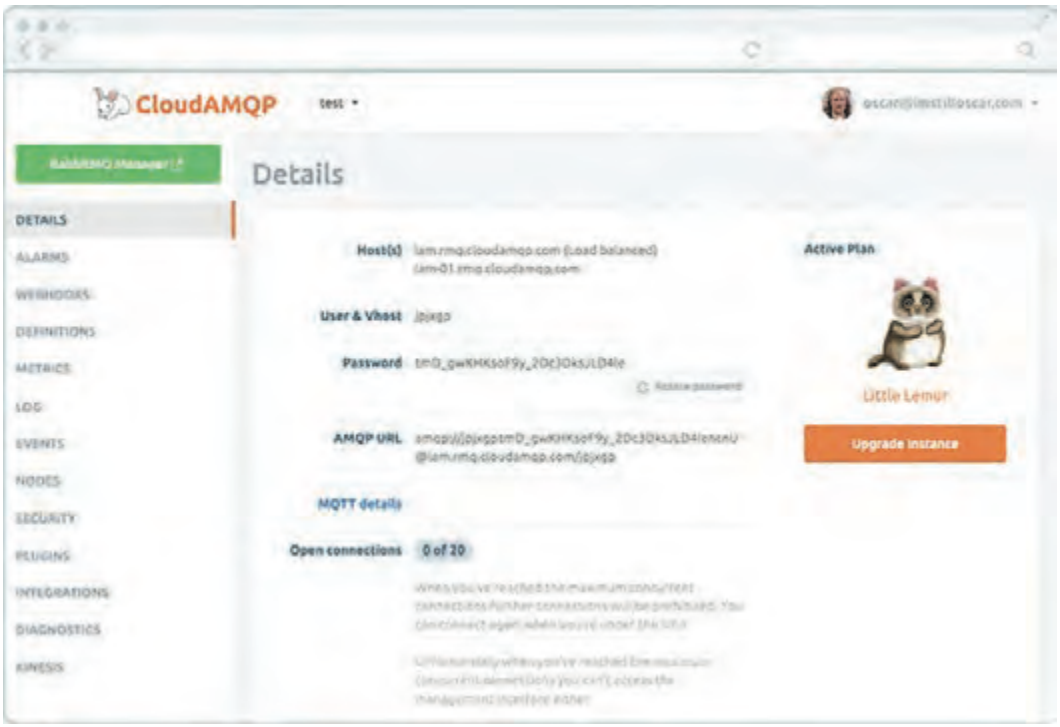


Figure 7 - Detailed information of an instance in the CloudAMQP Console.

The Management Interface - Management and Monitoring

RabbitMQ provides an easy to use web user interface (UI) for management and monitoring of your RabbitMQ server. A link to the management interface can be found on the details page for your CloudAMQP instance.

The management interface allows you to handle, create, delete and list queues. It also gives you the possibility to monitor queue length, check message rate, change and add users permissions, and much more.

Detailed information about the management interface will be given in the chapter: The Management Interface.

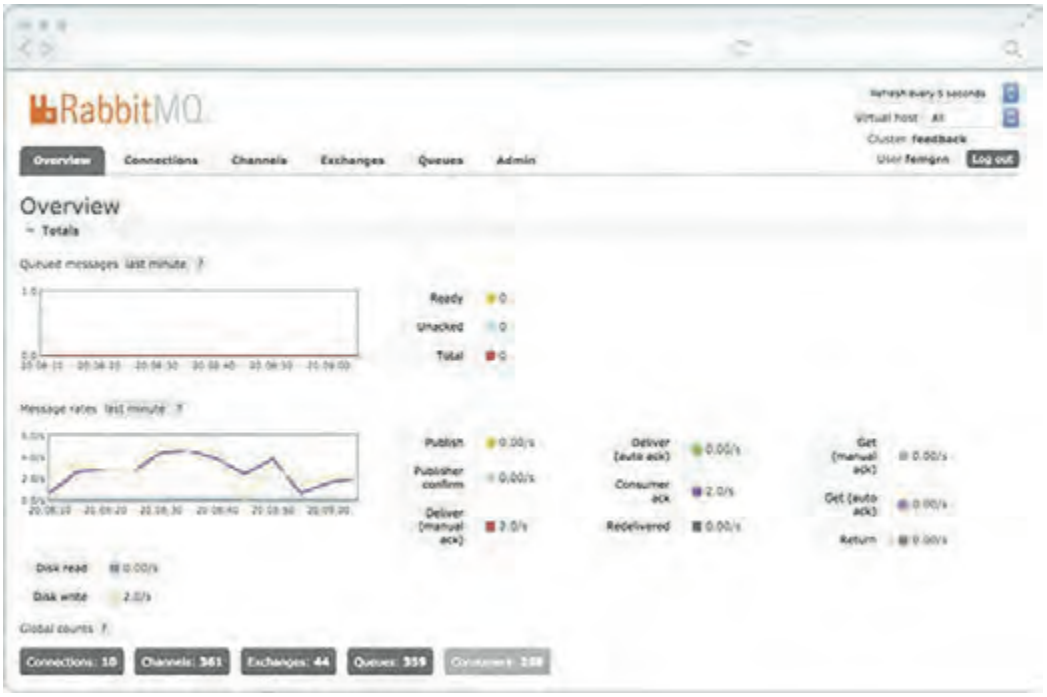


Figure 8 - The overview window in the RabbitMQ management interface.

Publish and consume messages

RabbitMQ speaks the AMQP protocol by default. To be able to communicate with RabbitMQ you need a library that understands the same protocol as RabbitMQ. You need to download the client library for the programming language that you intend to use for your applications. A client library is an application programming interface (API) to use when writing client applications. A client library has several methods that can be used, in this case, to communicate with RabbitMQ. The methods should be used when you, for example, connect to the RabbitMQ broker (using the given parameters, hostname, port number, etc.) or when you declare a queue or an exchange. There is a choice of libraries for all major programming languages.

Setting up a connection and publishing/consuming a message:

1. First of all, we need to set up/create a connection object. Here, the username, password, connection URL, port, etc., will be specified. A TCP connection will be set up between the application and RabbitMQ.

2. Secondly, a channel needs to be opened. The connection interface helps you accomplish that. You are now ready to send and receive messages.

3. Declare/create a queue. Declaring a queue will cause it to be created if it does not already exist. All queues need to be declared before they can be used.
4. Set up exchanges and bind a queue to an exchange. Messages are only routed to a queue if the queue is bound to an exchange.
5. Publish a message to an exchange, and consume a message from the queue.
6. Close the channel and the connection.

Sample code

Sample code for Ruby, Node.js, and Python can be found in upcoming chapters. It's possible to use different programming languages in different parts of your system. The subscriber could, for example, be written in Node.js while the subscriber is written in Python.

Hint: Separate your Projects and Environments Using Vhosts

You might create different databases within a single PostgreSQL host for different projects. In the same way, vhosts makes it possible to separate applications on one broker. You can isolate users, exchanges, queues etc to one specific vhost. You can separate environments, e.g., production to one vhost and staging to another vhost, within the same broker, instead of setting up multiple brokers. The downside of using a single instance is that there's no resource isolation between vhosts.

Shared plans on CloudAMQP are located on isolated vhosts.

Exchanges, routing keys and bindings

What are the exchanges, bindings and routing keys? In what way are exchanges and queues associated with each other? When should they be used and how? This chapter explains the different types of exchanges in RabbitMQ and the scenarios of when to use them.

As mentioned in the previous chapter - messages are not published directly to a queue. Instead, the producer sends messages to an exchange. Exchanges are the message routing agents, living in a virtual host within RabbitMQ. It accepts messages from the producer application and routes them to message queues with the help of header attributes, bindings, and routing keys.

A binding is a "link" that you configure to connect a queue to an exchange. The routing key is a message attribute. The exchange might look at this key (depending on exchange type) when deciding on how to route the message to the correct queue.

Exchanges, connections, and queues can be configured with properties such as durable, temporary, and auto delete upon creation. Durable exchanges survive server restarts and last until they are explicitly deleted. Temporary exchanges exist until RabbitMQ is shut down. Auto-deleted exchanges are removed once the last bound object is unbound from the exchange.

In RabbitMQ, four different types of exchanges route the message differently using different parameters and bindings setups. Clients can create their exchanges, or use the predefined default exchanges.

Direct Exchange

A direct exchange delivers messages to queues based on a message routing key. The routing key is a message attribute added to the message by the producer. The routing key can be seen as an "address" that the exchange uses to decide on how to route the message. **A message goes to the queue(s) whose binding key perfectly matches the routing key of the message.**

The direct exchange type is useful when you would like to distinguish messages published to the same exchange using a simple string identifier.

Queue A (create_pdf_queue) in figure 9 is bound to a direct exchange (pdf_events) with the binding key pdf_create. When a new message with routing key pdf_create arrives at the direct exchange, the exchange routes it to the queue where the binding_key = routing_key, in this case to queue A (create_pdf_queue).

Scenario 1

- Exchange: pdf_events
- Queue A: create_pdf_queue
- The binding key between exchange (pdf_events) and Queue A (create_pdf_queue): pdf_create

Scenario 2

- Exchange: pdf_events
- Queue B: pdf_log_queue
- Binding key between exchange (pdf_events) and Queue B (pdf_log_queue): pdf_log

Example

Example: A message with the routing key *pdf_log* is sent to the exchange *pdf_events* (*figure 9*). The message is routed to pdf_log_queue because of the routing key (*pdf_log*) matches the binding key (*pdf_log*).

Note: If the message routing key does not match any binding key, the message is discarded.

The default exchange AMQP brokers must provide for the direct exchange is "amq.direct".

Default exchange

The default exchange is a pre-declared direct exchange with no name, usually referred to with the empty string "". When you use the default exchange, your message is delivered to the queue with a name equal to the routing key of the message. Every queue is automatically bound to the default exchange with a routing key which is the same as the queue name.

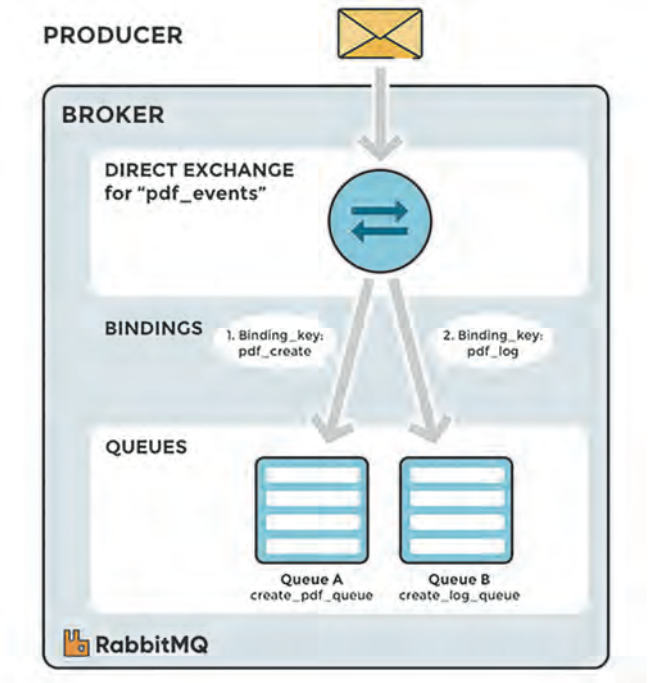


Figure 9 - A message is directed to the queue where the binding key is an exact match of the message's routing key.

Topic Exchange

Topic exchanges route messages to a queue based on a wildcard match between the routing key and something called the routing pattern, which is specified by the queue binding. Messages can be routed to one or many queues depending on this wildcard match.

The routing key must be a list of words, delimited by a period (.), examples can be *agreements.us* or *agreements.eu.stockholm*, which in this case identifies agreements that are set up for a company with offices in lots of different locations. The routing patterns may contain an asterisk ("*") to match a word in a specific position of the routing key (e.g., a routing pattern of *agreements.*.*.b.** only match routing keys where the first word is *agreements* and the fourth word is "b"). A pound symbol ("#") indicates a match on zero or more words (e.g., a routing pattern of *agreements.eu.berlin.#* matches any routing keys beginning with *agreements.eu.berlin*).

The consumers indicate which topics they are interested in (like subscribing to a feed of an individual tag). The consumer creates a queue and sets up a binding with a given routing pattern to the exchange. **All messages with a routing key that match the routing pattern are routed to the queue and stay there until the consumer consumes the message.**

The default exchange AMQP brokers must provide for the topic exchange is amq.topic.

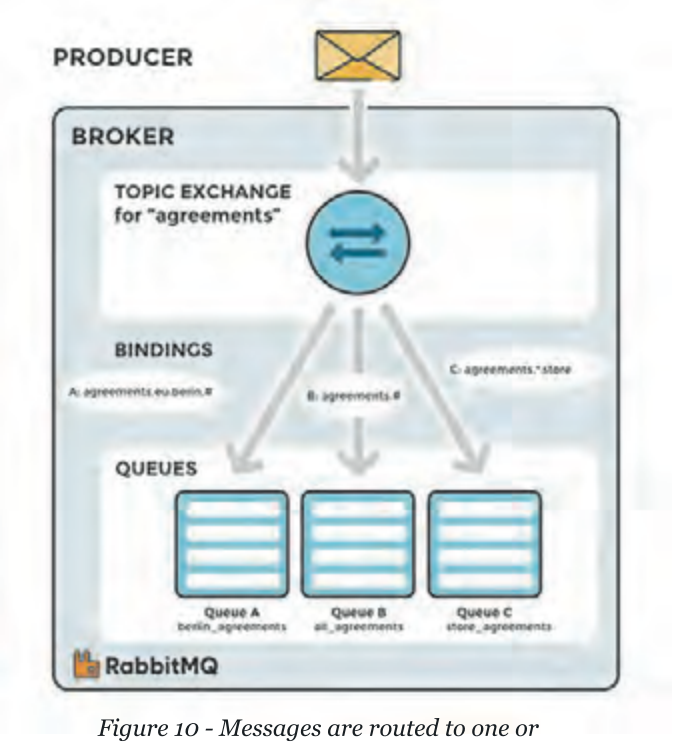


Figure 10 - Messages are routed to one or many queues based on a match between a message routing key and the routing patterns.

Scenario 1

Figure 10 shows an example where consumer A is interested in all the agreements in Berlin.

- Exchange: agreements
- Queue A: berlin_agreements
- Routing pattern between exchange (agreements) and Queue A (berlin_agreements): agreements.eu.berlin.#
- Example of message routing key that matches: agreements.eu.berlin and agreements.eu.berlin.store

Scenario 2

Consumer B is interested in all the agreements.

- Exchange: agreements

- Queue B: all_agreements
- Routing pattern between exchange (agreements) and Queue B (all_agreements): agreements.#
- Example of message routing key that matches: agreements.eu.berlin and agreements.us

Scenario 3

Consumer C is interested in all agreements for European stores.

- Exchange: agreements
- Queue C: store_agreements
- Routing pattern between exchange (agreements) and Queue C (store_agreements): agreements.eu.*.store
- Example of message routing keys that will match: agreements.eu.berlin.store and agreements.eu.stockholm.store

Example

A message with routing key agreements.eu.berlin is sent to the exchange *agreements*. The message is routed to the queue *berlin_agreements* because of the routing pattern of *agreements.eu.berlin.#* matches any routing keys beginning with *agreements.eu.berlin*. The message is also routed to the queue all_agreements since the routing key (*agreements.eu.berlin*) also matches the routing pattern *agreements.#*.

Fanout Exchange

The fanout exchange copies and routes a received message to all queues that are bound to it regardless of routing keys or pattern matching as with direct and topic exchanges. Keys provided will be ignored.

Fanout exchanges can be useful when the same message needs to be sent to one or more queues with consumers who may process the same message in different ways, like in a distributed systems where you need to broadcast various state and configuration updates.

Figure 11 shows an example where a message received by the exchange is copied and routed to all three queues that are bound to the exchange. It could be sport or weather news updates that should be sent out to each connected mobile device when something happens.

The default exchange AMQP brokers must provide for the fanout exchange is "amq.fanout".

Scenario 1

* Exchange: sport_news

* Queue A: Mobile client queue A

* Binding: Binding between the exchange (sport_news) and Queue A (Mobile client queue A)

Example

A message is sent to the exchange *sport_news (figure 11)*. The message is routed to all queues (Queue A, Queue B, Queue C) because all queues are bound to the exchange. Provided routing keys are ignored.

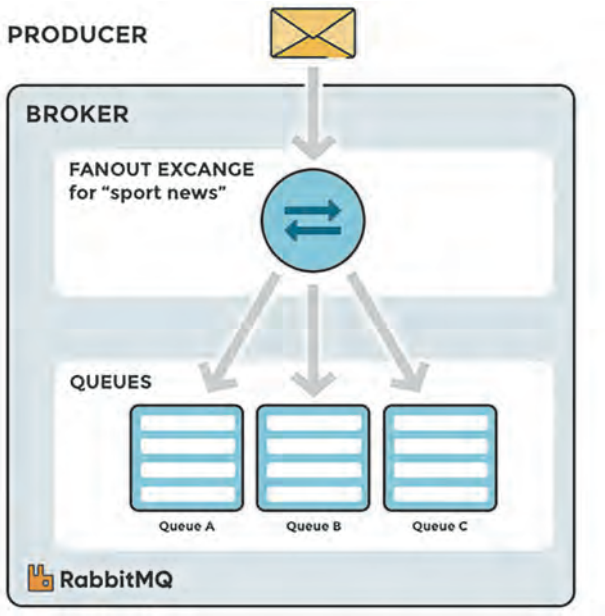


Figure 11 - Fanout Exchange: The received message is routed to all queues that are bound to the exchange.

Headers Exchange

The headers exchanges route their messages based on arguments containing headers and optional values. They are very similar to topic exchanges but decides their routes based on header values instead of routing keys. A message is considered to be matching if the value of the header equals the value specified on the binding.

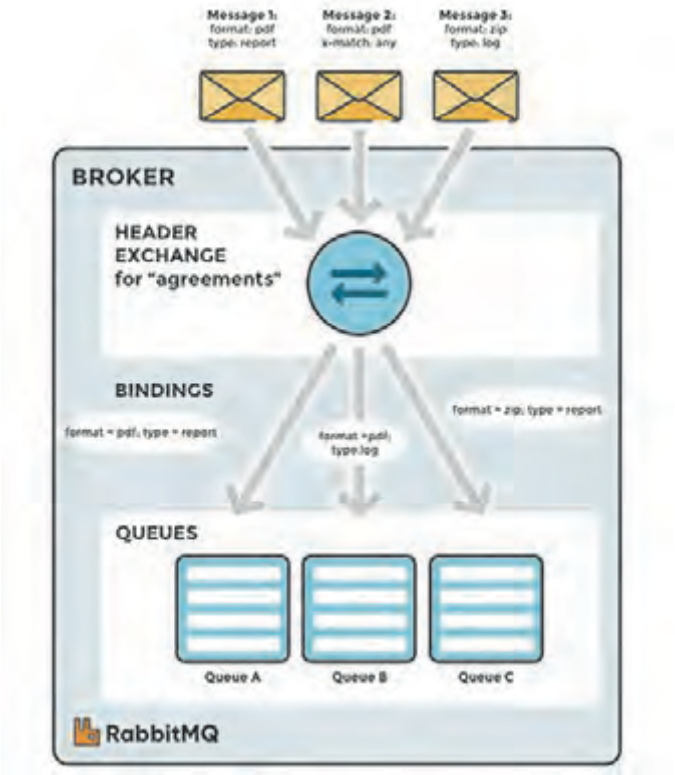


Figure 12 - Headers exchange routes messages to queues that are bound using arguments (key and value) containing headers and optional values.

A unique argument named "x-match", which can be added in the binding between your exchange and your queue, decides if all headers must match or just one. Either any common header between the message and the binding counts as a match, or all the headers referenced in the binding need to be present in the message for it to match. The "x-match" property can have two different values: "any" or "all", where "all" is the default value. A value of "all" means all header pairs (key, value) must match and a value of "any" means at least one of the header pairs must match. Headers can be constructed using a wider range of data types. Integers and hashes examples are good instead of the commonly used string. The headers exchange type (used with the binding argument "any") is useful for directing messages which may contain a subset of known (unordered) criteria.

The default exchange AMQP brokers must provide for the header exchange is "amq.headers".

- Exchange: Binding to Queue A with arguments (key = value): format = pdf, type = report, x-match = all
- Exchange: Binding to Queue B with arguments (key = value): format = pdf, type = log, x-match = any
- Exchange: Binding to Queue C with arguments (key = value): format = zip, type = report, x-match = all

Scenario 1

Message 1 is published to the exchange with the header arguments (key = value): "format = pdf", "type = report" and with the binding argument "x-match = all"

Message 1 is delivered to Queue A - all key/value pairs match

Scenario 2

Message 2 is published to the exchange with the header arguments (key = value): "format = pdf" and with the binding argument "x-match = any"

Message 2 is delivered to Queue A and Queue B - the queue is configured to match any of the headers (format or log).

Scenario 3

Message 3 is published to the exchange with the header arguments of (key = value): "format = zip", "type = log" and with the binding argument "x-match = all"

Message 3 is not delivered to any queue - the queue is configured to match all of the headers (format and log).

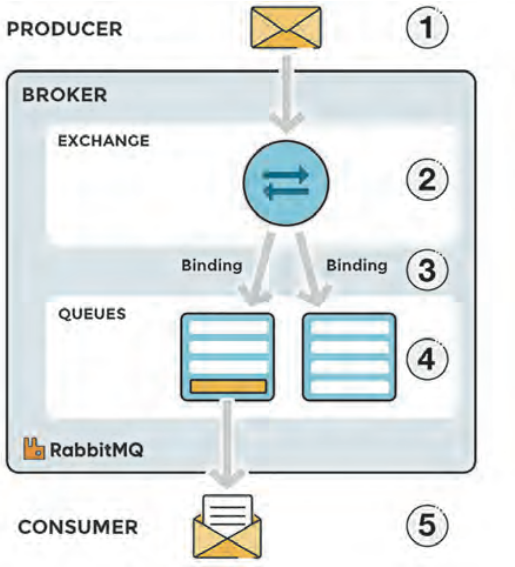
Dead Letter Exchange

The broker won't return an error and the message is silently dropped if you publish a message to a queue that hasn't been declared or to an exchange with a routing key that doesn't match any existing queues. RabbitMQ provides an AMQP extension known as the Dead Letter Exchange - it provides functionality to capture messages that are not deliverable.

RabbitMQ and client libraries

A client library that understands the same protocol as RabbitMQ is needed when you want to communicate with RabbitMQ. Developers have many options for AMQP client libraries in many different languages. Those client libraries have several methods that can be used to communicate with your RabbitMQ instance. This section of the book shows code examples for Ruby, Node.JS and Python.

The tutorial follows the scenario from the previous chapter; where a web application allows users to upload user information to a website. The application handles the data and generates a PDF and emails it back to the user. You need a RabbitMQ instance to get started with the coding examples.



RabbitMQ and Ruby with Bunny

Get started with RabbitMQ and Ruby.

Start by downloading the RabbitMQ client library for Ruby. Ruby developers have a number of options for AMQP client libraries. In this example, we will use Bunny, an asynchronous client for publishing and consuming messages.

When running the full example_publisher.rb code below, a connection is established between the RabbitMQ instance and your application. Queues and exchanges are declared and created if they do not already exist, and finally, a message is published.

The example_consumer.rb code sets up a connection and subscribes to a queue. The messages are handled one by one and sent to the PDF processing method.

First of all the full code for the publisher and the consumer is given, the same code is then divided into blocks and carefully explained.

```
# example_publisher.rb
require "rubygems"
require "bunny"
require "json"

# Returns a connection instance
conn = Bunny.new ENV['CLOUDAMQP_URL']
# The connection is established when start is called
conn.start

# create a channel in the TCP connection
ch = conn.create_channel

# Declare a queue with a given name, examplequeue.
# In this example is a durable shared queue used.
q  = ch.queue("examplequeue", :durable => true)

# Bind a queue to an exchange
```

```
x = ch.direct("example.exchange", :durable => true)

# A message is only routed to a queue if the exchange is bound to a queue
q.bind(x, :routing_key => "process")

information_message = "{\"mail\":\"e@m.com\",\"name\":\"n\",\"size\":\"s\"}"

x.publish(information_message,
  :timestamp      => Time.now.to_i,
  :routing_key    => "process"
)

sleep 1.0
conn.close
```

```
# example_consumer.rb
require "rubygems"
require "bunny"
require "json"

# Returns a connection instance
conn = Bunny.new ENV['CLOUDAMQP_URL']
#The connection is established when start is called
conn.start

# Create a channel in the TCP connection
ch = conn.create_channel
# Declare a queue with a given name, examplequeue.
# In this example is a durable shared queue used.
q  = ch.queue("examplequeue", :durable => true)

# Method for the PDF processing
def pdf_processing(json_information_message)
  puts "Handling pdf processing for: "
  puts json_information_message['email']
  sleep 5.0
  puts "pdf processing done"
end

# Set up the consumer to subscribe from the queue
q.subscribe(:block => false) do |delivery_info, properties, payload|
  json_information_message = JSON.parse(payload)
  pdf_processing(json_information_message)
end
```

Tutorial source code - Publisher

Set up a connection

```
#example_consumer.rb
require "rubygems"
require "bunny"
require "json"

# Returns a connection instance
conn = Bunny.new ENV['CLOUDAMQP_URL']

# The connection is established when start is called
conn.start
```

`Bunny.new` returns a connection instance. Use the CLOUDAMQP_URL as connection URL. It can be found on the details page for your CloudAMQP instance. The CLOUDAMQP_URL is a string including data for the instance, such as username, password, hostname, and vhost. The connection is established when start is called, `conn.start`

Create a channel

```
#Create a channel in the TCP connection
ch = conn.create_channel
```

A channel (a virtual connection) needs to be created in the TCP connection. When you are publishing or consuming messages or subscribing to a queue, it is all done over a channel.

Declare a queue

```
#Declare a queue with a given name
q  = ch.queue("examplequeue", :durable => true)
```

`ch.queue` is used to declare a queue with a particular name, in this case, the queue is called *examplequeue*. The queue in this example is marked as durable, meaning that RabbitMQ will not lose the queue during a RabbitMQ restart.

Bind the queue to an exchange

```
#For messages to be routed to queues, queues need to be bound to exchanges.
x = ch.direct("example.exchange", :durable => true)

#Bind a queue to an exchange
q.bind(x, :routing_key => "process")
```

A direct exchange is used, it delivers messages to queues based on a message *routing key*. The routing key "process" is used for this binding. The exchange is first created and then bound to the queue.

Publish a message

```
information_message = "{\"email\": \"e@m.com\",\"name\": \"n\",\"size\": \"s\
"}"

  x.publish(information_message,
    :timestamp      => Time.now.to_i,
    :routing_key    => "process"
  )
```

`information_message` is all the information that is sent to the exchange. The direct exchanges use the message routing key for routing, meaning the producers need to specify the routing key in the message for it to not get dropped.

Close the connection

```
sleep 1.0
conn.close
```

`conn.close` automatically close all channels on the connection.

Tutorial source code - Consumer

```
#Method for the pdf processing
def pdf_processing(json_information_message)
  puts "Handling pdf processing for: "
  puts json_information_message['email']
  sleep 5.0
  puts "pdf processing done"
end
```

The method `pdf_processing` is a method stub that sleeps for 5 seconds to simulate the pdf processing.

Set up the consumer

```
#Set up the consumer to subscribe from the queue
q.subscribe(:block => false) do |delivery_info, properties, payload|
  json_information_message = JSON.parse(payload)
  pdf_processing(json_information_message)
end
```

`subscribe` consumes messages and processes them. It will be called every time a message arrives. The subscribe method will not block the calling thread by default.

More information about Ruby and CloudAMQP can be found on the CloudAMQP web page.

RabbitMQ and Node.js with Amqplib

How to get started with RabbitMQ and Node.js.

Start by downloading the client library for Node.js. Node developers have a number of options for AMQP client libraries. In this example, amqplib is used. Continue on by adding `amqplib` as a dependency to your `package.json` file.

When running the full code given, a connection is established between the RabbitMQ instance and your application. Queues and exchanges are declared and created if they do not already exist and finally, a message is published. The publish method queue messages internally if the connection is down and resend them later. The consumer subscribes to the queue. Messages are handled one by one and sent to the PDF processing method.

A new message is published every second. A default exchange, identified by the empty string ("") is used. The default exchange means that messages are routed to the queue with the name specified by `routing_key`, if it exists. (The default exchange is a direct exchange with no name)

The full code can be downloaded from CloudAMQP documentation pages.

Tutorial source code

Load amqplib

```
// Access the callback-based API
var amqp = require('amqplib/callback_api');
var amqpConn = null;
```

Set up a connection

```
function start() {
  amqp.connect(process.env.CLOUDAMQP_URL + "?heartbeat=60", function(err, con
n) {
    if (err) {
```

```
      console.error("[AMQP]", err.message);
      return setTimeout(start, 1000);
    }
    conn.on("error", function(err) {
      if (err.message !== "Connection closing") {
        console.error("[AMQP] conn error", err.message);
      }
    });
    conn.on("close", function() {
      console.error("[AMQP] reconnecting");
      return setTimeout(start, 1000);
    });

    console.log("[AMQP] connected");
    amqpConn = conn;

    whenConnected();
  });
}
```

The `start` function will establish a connection to RabbitMQ. If the connection is closed or fails to be established, it will try to reconnect.

`amqpConn` will hold the connection and channels will be set up in the connection.

`whenConnected` will be called when a connection is established.

```
function whenConnected() {
  startPublisher();
  startWorker();
}
```

The function `whenConnected` calls two functions, one function that starts the publisher and one that starts the worker (the consumer).

Start the publisher

```
var pubChannel = null;
var offlinePubQueue = [];
function startPublisher() {
  amqpConn.createConfirmChannel(function(err, ch) {
    if (closeOnErr(err)) return;
    ch.on("error", function(err) {
      console.error("[AMQP] channel error", err.message);
    });
    ch.on("close", function() {
```

```
      console.log("[AMQP] channel closed");
    });

    pubChannel = ch;
    while (true) {
      var [exchange, routingKey, content] = offlinePubQueue.shift();
      publish(exchange, routingKey, content);
    }
  });
}
```

`createConfirmChannel` opens a channel which uses "confirmation mode". A channel in confirmation mode requires each published message to be 'acked' or 'nacked' by the server, thereby indicating that it has been handled.

`offlinePubQueue` is an internal queue for messages that could not be sent when the application was offline. The application will keep an eye on this queue and try to resend any messages added to it.

Publish

```
function publish(exchange, routingKey, content) {
  try {
    pubChannel.publish(exchange, routingKey, content, { persistent: true },
        function(err, ok) {
          if (err) {
            console.error("[AMQP] publish", err);
            offlinePubQueue.push([exchange, routingKey, content]);
            pubChannel.connection.close();
          }
        });
  } catch (e) {
    console.error("[AMQP] publish", e.message);
    offlinePubQueue.push([exchange, routingKey, content]);
  }
}
```

The `publish` function will publish a message to an exchange with a given routing key. If an error occurs the message will be added to the internal queue, `offlinePubQueue`

Consumer

```
# A worker that acks messages only if processed successfully
function startWorker() {
  amqpConn.createChannel(function(err, ch) {
    if (closeOnErr(err)) return;
```

```
    ch.on("error", function(err) {
      console.error("[AMQP] channel error", err.message);
    });
    ch.on("close", function() {
      console.log("[AMQP] channel closed");
    });
    ch.prefetch(10);
    ch.assertQueue("jobs", { durable: true }, function(err, _ok) {
      if (closeOnErr(err)) return;
      ch.consume("jobs", processMsg, { noAck: false });
      console.log("Worker is started");
    });

    function processMsg(msg) {
      work(msg, function(ok) {
        try {
          if (ok)
            ch.ack(msg);
          else
            ch.reject(msg, true);
        } catch (e) {
          closeOnErr(e);
        }
      });
    }
  });
}
```

`amqpConn.createChannel` creates a channel on the connection. `ch.assertQueue` creates a queue if it does not already exist. `ch.consume` sets up a consumer with a callback to be invoked with each message it receives. The function called for each message is called `processMsg`

`processMsg` processes the message. It will call the work function and wait for it to finish.

```
function work(msg, cb) {
  console.log("PDF processing of ", msg.content.toString());
  cb(true);
}
```

The `work` function includes the handling of the message information and the creation of the PDF.

Close the connection on error

```
function closeOnErr(err) {
  if (!err) return false;
  console.error("[AMQP] error", err);
  amqpConn.close();
  return true;
}
```

Publish

```
setInterval(function() {
  publish("", "jobs", new Buffer("work work work"));
}, 1000);

start();
```

A new message is published every second. A default exchange, identified by the empty string ("") is used. The default exchange means that messages are routed to the queue with the name specified by routing_key, if it exists. (The default exchange is a direct exchange with no name.)

More information about Node.js and CloudAMQP can be found on the CloudAMQP web page.

RabbitMQ and Python with Pika

This chapter explains how to get started with RabbitMQ using Python with the client library Pika.

Start by downloading the client library for Python. The recommended library for Python is Pika. Put `pika==0.9.14` in your `requirement.txt` file.

When running the full code given below, a connection is established between the RabbitMQ instance and your application. Queues and exchanges will be declared and created if they do not already exist and finally, a message is published to the message queue. The consumer subscribes to the queue, and messages are handled one by one and sent to the PDF processing method.

A default exchange, identified by the empty string ("") is used. By using the default exchange, messages are routed to the queue with the name matching the routing_key, if it exists.

Full code

```
# example_publisher.py
import pika, os, logging
logging.basicConfig()

# Parse CLODUAMQP_URL (fallback to localhost)
url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost/%2f')
params = pika.URLParameters(url)
params.socket_timeout = 5

# Connect to CloudAMQP
connection = pika.BlockingConnection(params)
# start a channel
channel = connection.channel()
# Declare a queue
channel.queue_declare(queue='pdfprocess')
```

```
# send a message
channel.basic_publish(exchange='', routing_key='pdfprocess', body='User infor
mation')
print " [x] Message sent to consumer"
connection.close()
```

```
# example_consumer.py
import pika, os, logging, time
logging.basicConfig()

def pdf_process_function(msg):
  print "PDF processing"
  time.sleep(5) # delays for 5 seconds
  print "PDF processing finished";
  return;

# Parse CLODUAMQP_URL (fallback to localhost)
url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost/%2f')
params = pika.URLParameters(url)
params.socket_timeout = 5
# Connect to CloudAMQP
connection = pika.BlockingConnection(params)
channel = connection.channel() # start a channel

# create a function which is called on incoming messages
def callback(ch, method, properties, body):
  pdf_process_function(body)

#set up subscription on the queue
channel.basic_consume(callback,
  queue='pdfprocess',
  no_ack=True)

# start consuming (blocks)
channel.start_consuming()
connection.close()
```

Tutorial source code - Publisher

```
# example_consumer.py
import pika, os, logging

# Parse CLODUAMQP_URL (fallback to localhost)
url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost/%2f')
params = pika.URLParameters(url)
params.socket_timeout = 5
```

Load the client library and set up some configuration parameters. The `DEFAULT_SOCKET_TIMEOUT` is set to 0.25s, we would recommend raising this to about 5s to avoid connection timeouts, `params.socket_timeout = 5` Other connection parameter options for Pika can be found here: http://pika.readthedocs.org/en/latest/modules/parameters.html

Set up a connection

```
# Connect to CloudAMQP
connection = pika.BlockingConnection(params)
```

`pika.BlockingConnection` establishes a connection with the RabbitMQ server.

Start a channel

```
channel = connection.channel()
```

`connection.channel` creates a channel over the TCP connection.

Declare a queue

```
# Declare a queue
channel.queue_declare(queue='pdfprocess')
```

`channel.queue_declare` creates a queue to which the message will be delivered. The queue is given the name *pdfprocess*.

Publish a message

```
channel.basic_publish(exchange='', routing_key='pdfprocess', body='User infor
mation')
print " [x] Message sent to consumer"
```

`channel.basic_publish` publishes a message through the channel. A default exchange, identified by the empty string ("") is used. The default exchange means that messages are routed to the queue with the name matching the messages routing_key if it has one.

Close the connection

```
connection.close()
```

The connection will be closed after the message has been published.

Consumer

Worker function

```
def pdf_process_function(msg):
  print "PDF processing"
  time.sleep(5) # delays for 5 seconds
  print "PDF processing finished";
  return;
```

The `pdf_process_function` sleeps for 5 seconds to emulate a PDF being created.

Function called for incoming messages

```
# create a function which is called on incoming messages
def callback(ch, method, properties, body):
  pdf_process_function(body)
```

The `callback` function is called once per message published to the queue. That function will in turn call a worker function that simulates the PDF-processing.

```
#set up subscription on the queue
channel.basic_consume(callback,
  queue='pdfprocess',
  no_ack=False)
```

`basic_consume` binds messages to the consumer callback function.

```


# start consuming (blocks)
channel.start_consuming()
connection.close()
```

`start_consuming` starts to consume messages from the queue.

More information about Python and CloudAMQP can be found on the CloudAMQP web page. We recommend you to check the CloudAMQP web page for recommendations if you are using the Celery task queue.

The Management Interface

The RabbitMQ management is a user-friendly interface that lets you monitor and handle your RabbitMQ server from a web browser. This section covers the different views that you can find in the RabbitMQ management interface.

Queues, connections, channels, exchanges, users and user permissions can all be handled - created, deleted and listed from the browser. You can monitor message rates and send/receive messages manually. RabbitMQ management is a plugin to RabbitMQ. It consists of a single static HTML page that makes background queries to the HTTP API for RabbitMQ. The management interface can be useful when you are debugging your applications or when you need an overview of the whole system.

The RabbitMQ management interface is enabled by default in CloudAMQP. A link can be found on the details page for your CloudAMQP instance.

All the tabs in the management menu are explained in this chapter. Screenshots from the views are shown for: overview, connections, and channels, exchanges, queues, and admin - users and permissions. A simple example will also show how to set up a queue with an exchange and add a binding between them.

Concepts

- **Cluster:** A cluster consists of a set of connected computers that work together. A RabbitMQ instance consisting of more than one node is called a RabbitMQ cluster. A cluster is a group of nodes, i.e., a group of computers.

- **Node:** A node is a single computer in the RabbitMQ cluster.

Overview

The overview, seen in figure 14, shows two graphs. One graph for queued messages and one with the message rate. You can change the time interval shown in the graph by pressing the text last minute above the graph. Information about all different statuses for messages can be found by pressing "?".

Queued messages

A graph of the total number of queued messages for all your queues. **Ready** shows the number of messages that are available to be delivered. **Unacked** are the number of messages for which the server is waiting for acknowledgment.

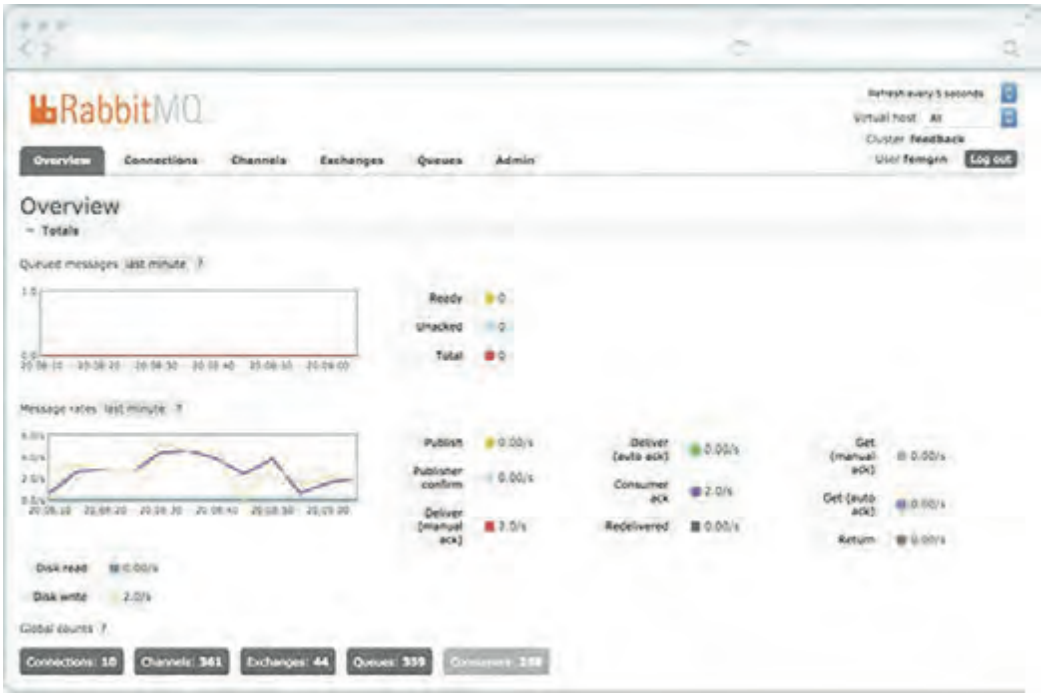


Figure 14 - The RabbitMQ management interface.

Message rate

A graph with the rate of how fast the messages are handled. **Publish** shows the rate at which messages are entering the server and **Confirm** shows a rate at which the server is confirming.

Global Count

This represents the total number of connections, channels, exchanges, queues and consumers for ALL virtual hosts the current user has access to.

Nodes

Nodes show information about the different nodes in the RabbitMQ cluster or information about one single node if only a single node is used. Here you can find information about server memory, the number of Erlang processes per node, and other node-specific information here. Info shows further information about the node and enabled plugins.

Figure 15 - Node-specific information.

Import/export definitions

It's possible to import and export configuration definitions. When you download the definitions, you get a JSON representation of your broker (your RabbitMQ settings). This can be used to restore exchanges, queues, virtual hosts, policies, and users. This feature can be used as a backup. Every time you make a change in the config, you can keep the old settings, just in case.

Figure 16 - Import/export definitions as a JSON file.

Connections and channels

RabbitMQ connections and channels can be in different states; starting, tuning, opening, running, flow, blocking, blocked, closing, closed. If a connection enters flow-control this often means that the client is being rate-limited in some way.

Connections

The connections tab *(figure 17)* shows the connections established to the RabbitMQ server. **vhost** shows in which vhost the connection operates and **username** shows the user associated with the connection. **Channels** will tell you the number of channels using the connection. **SSL/TLS** indicate whether the connection is secured with SSL or not.

If you click on one of the connections, you get an overview of that specific connection *(figure 18)*. You can view channels within the connection and its data rates. You can see client properties, and you can close the connection.

More information about the attributes associated with a can be found in the manual page for rabbitmqctl, the command line tool for managing a RabbitMQ broker.

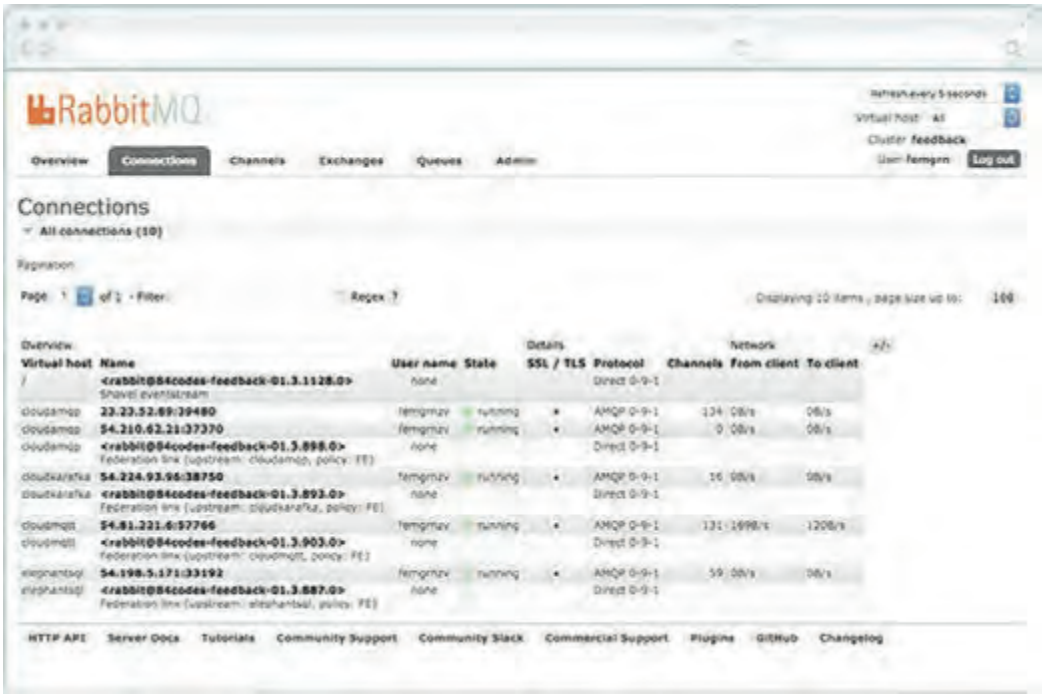


Figure 17 - The Connections tab in the RabbitMQ management interface.

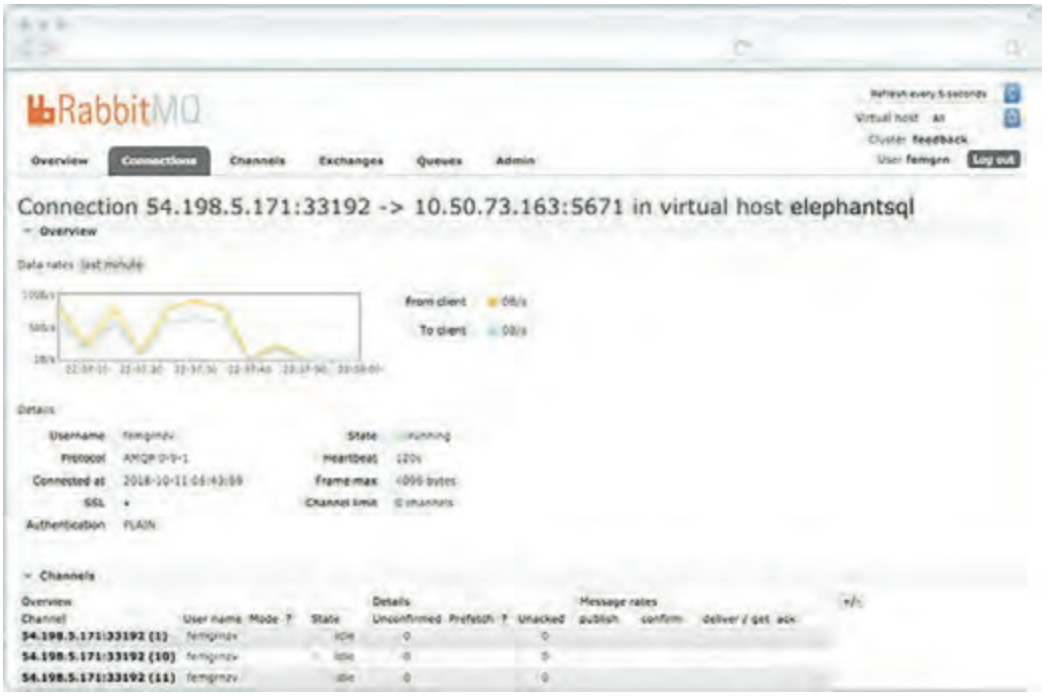


Figure 18 - Connection information for a specific connection.

Channels

The channel tab *(figure 19)* shows information about all the current channels. The vhost shows in which vhost the channel operates and the username shows the user associated with the channel. The guarantee mode can be in confirm or transactional mode. When a channel is in confirm mode, both the broker and the client count messages. The broker then confirms messages as it handles them. Confirm mode is activated once the *confirm.select* method is used on a channel.

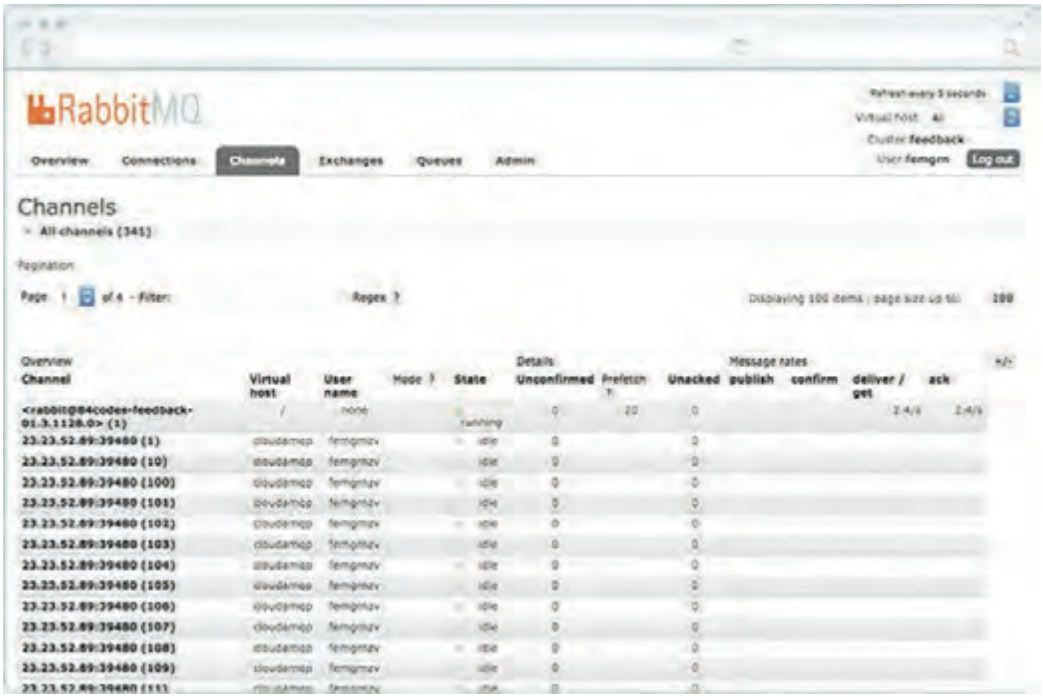


Figure 19 - The Channels tab with information about all the current channels.

If you click on one of the channels, you get a detailed overview of that specific channel *(figure 20)*. From here you can see the message rate and the number of logical consumers retrieving messages from the channel.

More information about the attributes associated with a channel can be found in the manual page for rabbitmqctl, the command line tool for managing a RabbitMQ broker.

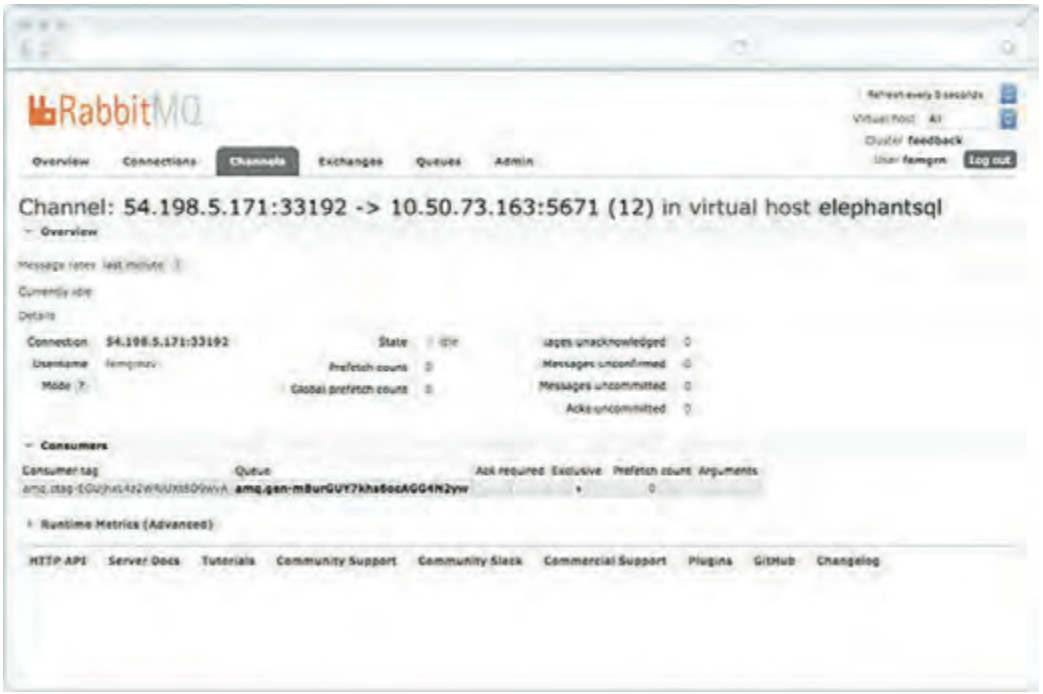


Figure 20 - Detailed information about a specific channel.

Exchanges

All exchanges can be listed from the exchange tab *(figure 21)*. **Virtual host** shows the vhost for the exchange, **type** is the exchange type such as direct, topic, headers and fanout. **Features** show the parameters for the exchange (e.g D stands for durable, and AD for auto-delete). Features and types can be specified when the exchange is created. In this list, there are some amq.* exchanges and the default (unnamed) exchange. These are created by default.

By clicking on the exchange name, a detailed page about the exchange is shown *(figure 21)*. You can add bindings to the exchange and view already existing bindings. You can also publish a message to the exchange or delete the exchange.

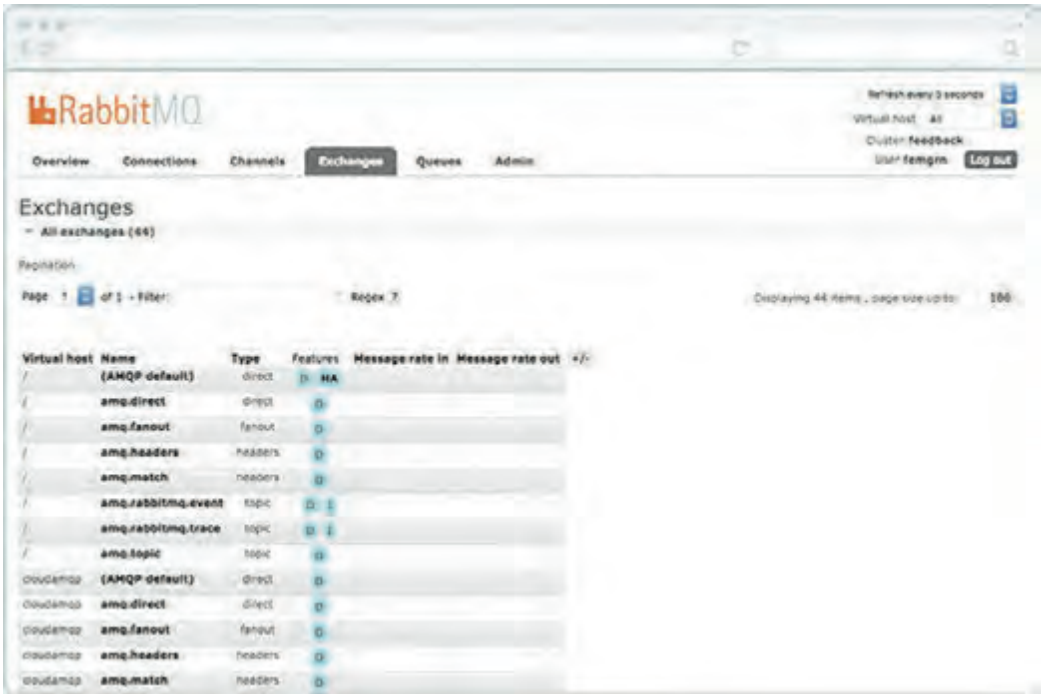


Figure 21 - The exchanges tab in the RabbitMQ management interface.

Figure 22 - Detailed view of an exchange.

Queues

The queue tab shows the queues for all or one selected vhost *(figure 23)*.

Queues have different parameters and arguments depending on how they were created. The features column show the parameters that belong to the queue. It could be features like:

- **Durable queue:** A durable queue ensures that RabbitMQ never loses the queue.
- **Message TTL:** The time a message published to a queue can live before it's discarded.
- **Auto-expire:** The time a queue can be unused before it's automatically deleted.
- **Max length:** How many (ready) messages a queue can hold before it starts to drop them.
- **Max length bytes:** The total body size of ready messages a queue can contain before it starts to drop them.

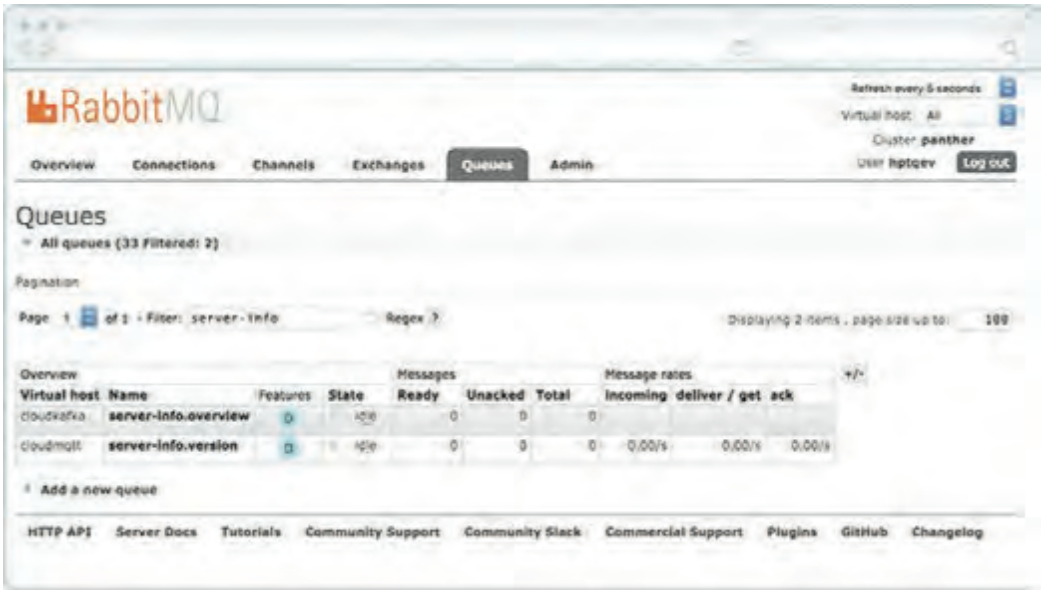


Figure 23 - The queues tab in the RabbitMQ management interface.

You can also create queues from this view.

If you click on any chosen queue from the list of queues, you will see all information about it, as seen in *figure 24*.

The first two graphs include the same information as the overview, but instead, they just show the number of queued messages and the message rates for this specific queue.

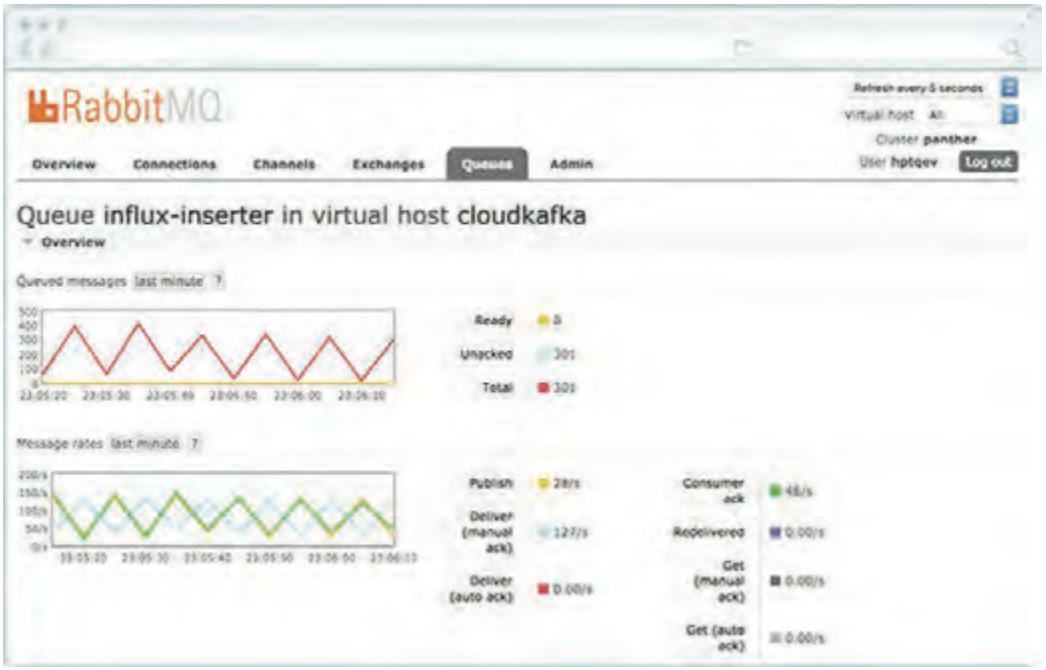


Figure 24 - Specific information about a single queue.

Consumers

The consumer's field shows the consumers/channels that are connected to the queue.

Figure 25 - Consumers connected to a queue.

Bindings

All active bindings to the queue are shown under bindings. You can also create a new binding to a queue from here or unbind a queue from an exchange *(figure 26)*.

Publish message

It's possible to manually publish a message to the queue from "publish message". The message will be published to the default exchange with the queue name as its routing key - ensuring that the message will be sent to the queue. It's also possible to publish a message to an exchange from the exchange view.

Figure 26 - The bindings interface.

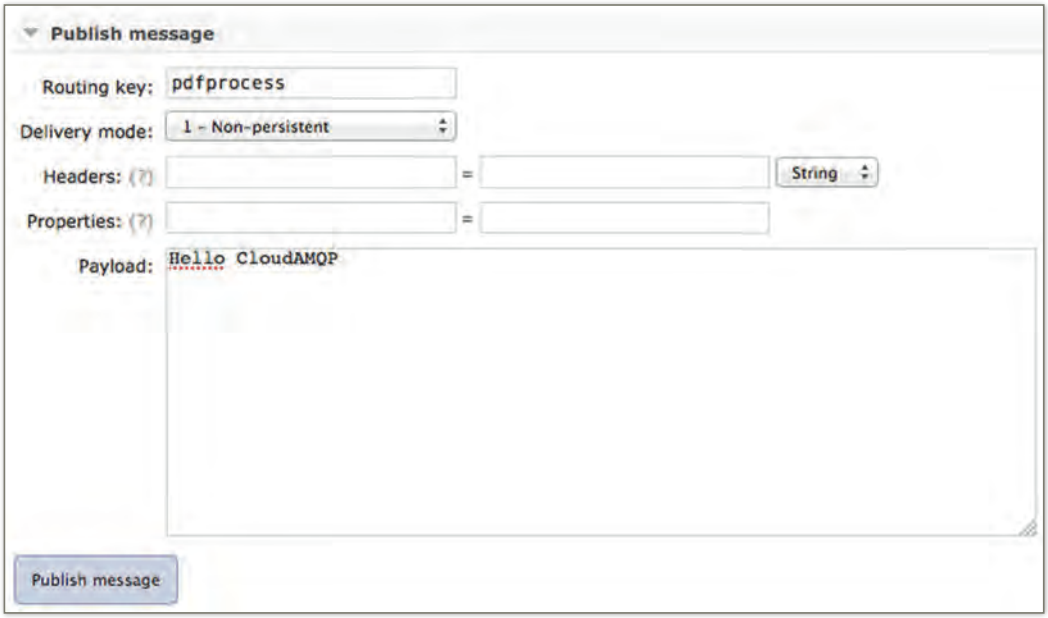


Figure 27 - Manually publishing a message to the queue.

Get message

It's possible to manually inspect the message in the queue. Get message gets the message to you, and if you mark it as requeue RabbitMQ puts it back in the queue in the same order.

Figure 28 - Manually inspect a message in the

Delete or Purge queue

A queue can be deleted by pressing the delete button. You can also empty the queue by pressing purge.

Figure 29 - Delete or purge a queue from the web interface.

Admin

From the Admin view *(figure 30)*, it's possible to add users and change user permissions. You can set up vhosts *(figure 31)*, policies, federation, and shovels. Information about shovels can be found here: https://www.rabbitmq.com/shovel.html. More information about federation will be given in part 2 of this book.

The example in *figure 32*, shows how you can create a queue example-queue and an exchange called *example.exchange (figure 33)*.

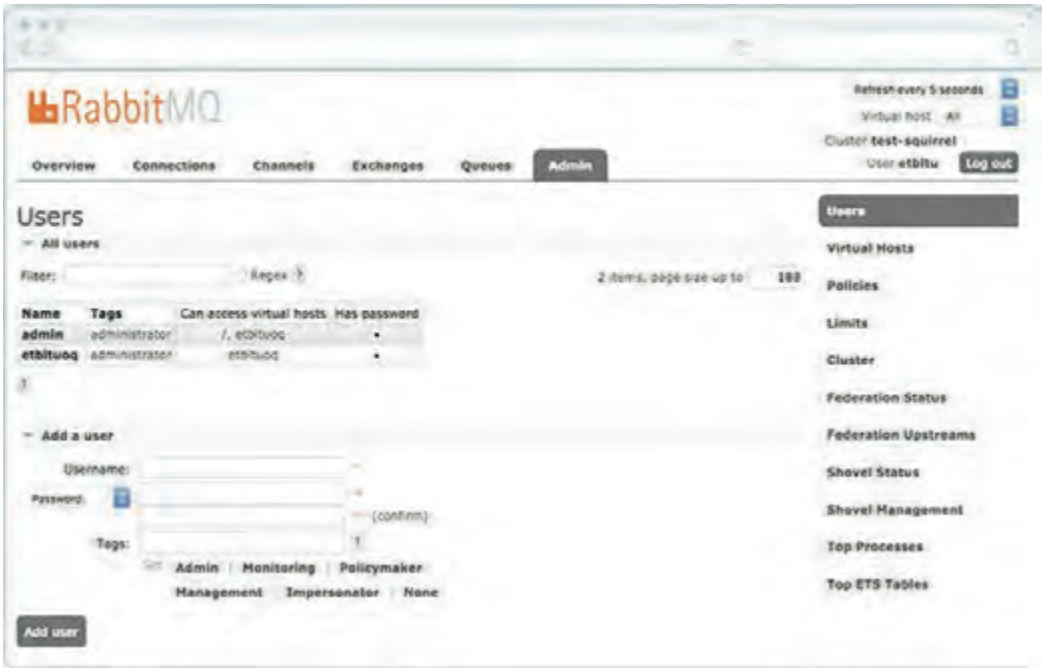


Figure 30 - The Admin interface where users can be added.

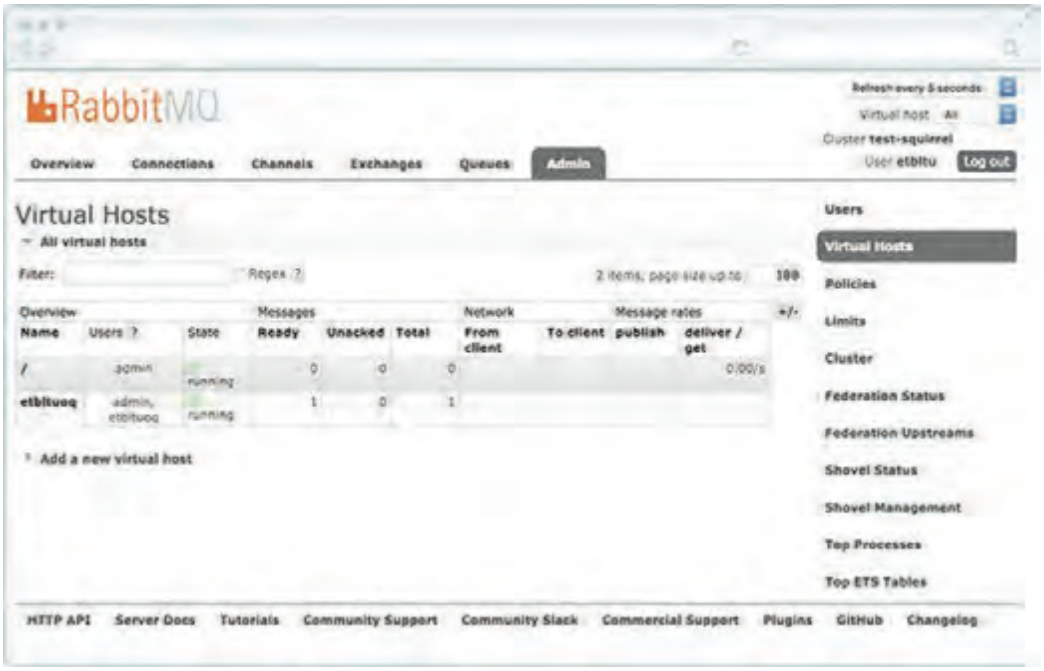


Figure 31 - Virtual Hosts can be added from the Admin tab.

The exchange and the queue are connected by a binding called pdfprocess *(figure 34)*. Messages published *(figure 35)* to the exchange with the routing key pdfprocess will end up in the queue *(figure 36)*.

A lot of things can be viewed and handled via the management interface, and it will give you a good overview of your system. By looking into the management interface, you will get a good understanding of RabbitMQ and how everything is related.

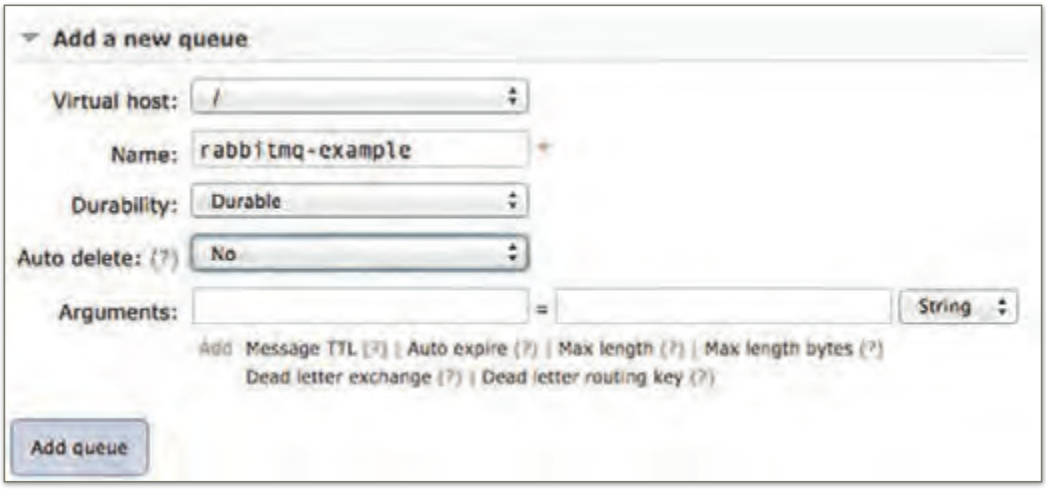


Figure 32 - Queue view, add queue.

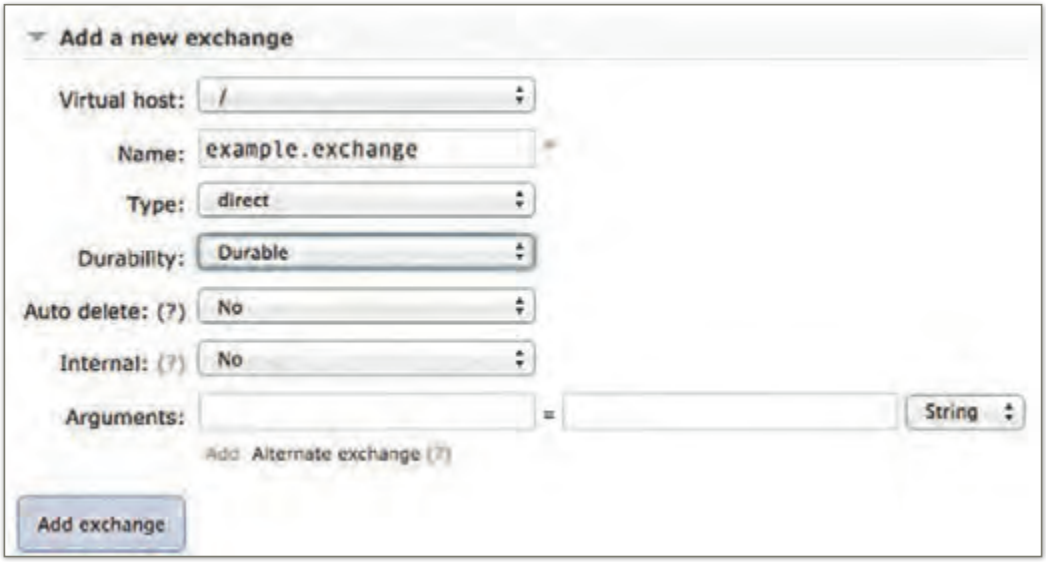


Figure 33 - Exchange view, add exchange.

Figure 34 - Click on the exchange or on the queue, go to "Add binding from this exchange" or "Add binding to this queue".

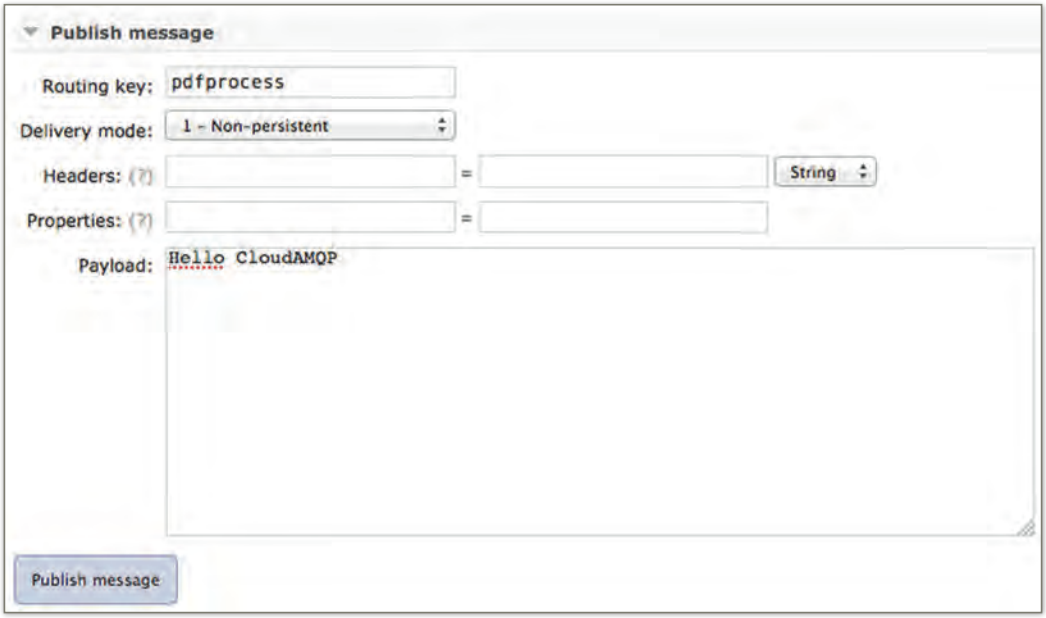


Figure 35 - Publish a message to the exchange with the routing key "pdfprocess".

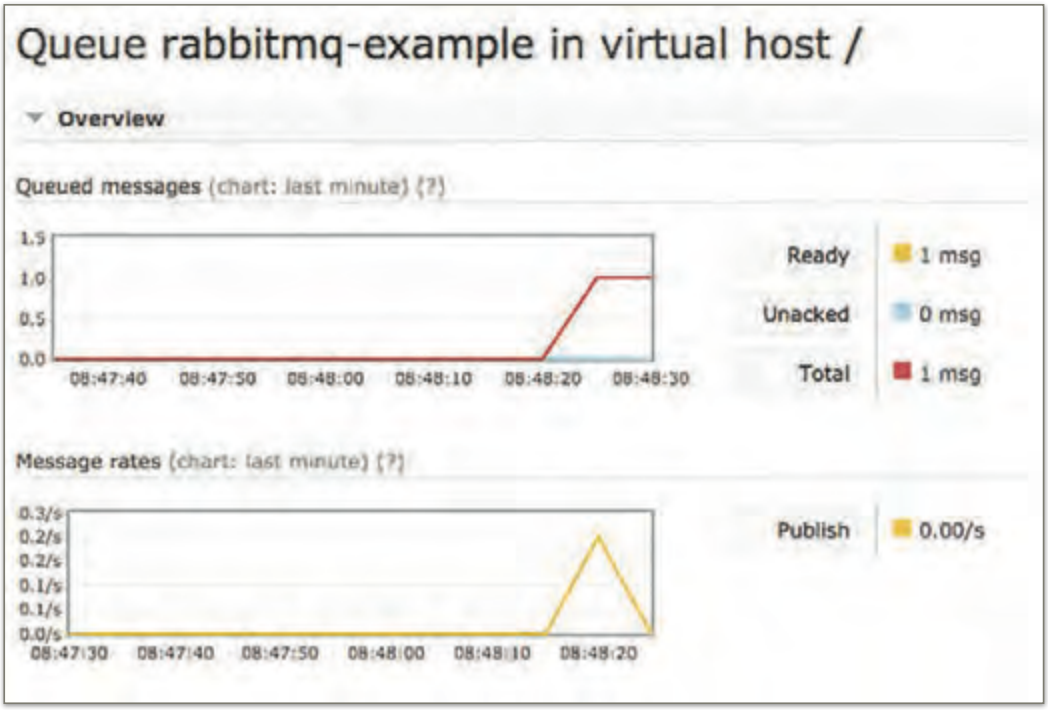


Figure 36 - Queue overview for example-queue when a message is published.

Part 2: Advanced Message Queuing

Some applications require high throughput while other applications are publishing batch jobs that can be delayed for a while. Tradeoffs have to be done between performance and guaranteed message delivery. The goal when designing your system should be to maximize combinations of performance and availability that make sense for your specific application. Bad architectural design decisions and client-side bugs can damage your broker or affect your throughput.

Part 2 of this book talks about the dos and the don'ts mixed with best practice for two different usage categories; high availability and high performance (high throughput). This part also mentions features in RabbitMQ that we at CloudAMQP would like to give some extra attention, for example, how to migrate your cluster with queue federation.

RabbitMQ Best Practice

How should you configure your RabbitMQ cluster for optimal performance and how should you configure it to get the most stable cluster? Here are RabbitMQ Best Practice recommendations, based on the experience CloudAMQP have gained while working with RabbitMQ. Queue size, common mistakes, lazy queues, prefetch values, connections and channels, HiPE and number of nodes in a cluster are some of the things discussed.

Queues

Let's start by optimizing queues.

Keep your queue short if possible

If it suits your use case, try keeping queues as short as possible. A message published to an empty queue will go straight out to the consumer, as soon as the queue receives the message (a persistent message in a durable queue will also go to disk). We recommend less than 10 000 messages in one queue.

Many messages in a queue can put a heavy load on RAM usage. In order to free up RAM, RabbitMQ starts flushing (page out) messages to disk. The page out process usually takes time and blocks the queue from processing messages when there are many messages to page out. A large amount of messages might have a negative impact on the broker since the process deteriorates queuing speed.

Note: Don't have too large queues

Short queues are the fastest. A message in an empty queue will go straight out to the consumer, as soon as the queue receives the message.

It's also time-consuming to restart a cluster with many messages since the index has to be rebuilt. It takes time to sync messages between nodes in the cluster after a restart.

Enable lazy queues to get predictable performance

A feature called lazy queues was added in RabbitMQ 3.6. Lazy queues write messages to disk immediately, thus spreading the work out over time instead of taking the risk of a performance hit somewhere down the road, as mentioned in the section above. This gives you a more predictable and smooth performance curve without any sudden drops, but at the cost of a little overhead. Messages are only loaded into memory when they are needed, and thereby the RAM usage is minimized, but the throughput time will be longer with lazy queues.

If you are sending many messages at once (e.g., processing batch jobs) or if you think that your consumers will not keep up with the speed of the publishers all the time, we recommend you to enable lazy queues.

Please note that you should disable lazy queues if you require high performance - if your queues always are short, or you have got a max-length policy.

Note: Use lazy queues to get predictable performance

We recommend using lazy queues when you know that you will have large queues from time to time, or when you are publishing *batch jobs*.

Limit queue size with TTL or max-length

Another thing that sometimes is recommended for applications that often get hit by spikes of messages, and where throughput is more important than anything else, is to set a max-length on the queue. This keeps the queue short by discarding messages from the head of the queue so that it's never getting larger than the max-length setting.

Number of queues

Queues are single-threaded in RabbitMQ, and one queue can handle up to about 50k messages/s. You achieve better throughput on a multi-core system if you have multiple queues and consumers. You achieve optimal throughput if you have as many queues as cores on the underlying node(s).

The RabbitMQ management interface collects and calculates metrics for every queue in the cluster. This might slow down the server if you have thousands upon thousands of active queues and consumers. The CPU and RAM usage may also be affected negatively if you have too many queues.

Note: Use multiple queues and consumers

You achieve optimal throughput if you have as many queues as cores on the underlying node(s).

Split your queues over different cores

Queue performance is limited to one CPU core (hardware thread) since a queue is single threaded. You will, therefore, get better performance if you split your queues over different cores, and also into different nodes if you have a RabbitMQ cluster. Routing messages to multiple queues, especially on different machines within the same cluster, can give a much higher overall performance.

RabbitMQ queues are bound to the node where they first were declared. All messages routed to a specific queue will end up on the node where that queue resides. You can manually split your queues evenly between nodes, but the downside is that you need to remember where your queue is located.

We recommend two plugins which can help you if you have multiple nodes or a single node cluster with multiple cores.

The consistent hash exchange plugin

The consistent hash exchange plugin allows you to use an exchange to load-balance messages between queues. Messages sent to the exchange are consistently and equally distributed across many queues, based on the routing key of the message. The plugin creates a hash of the routing key and spread the messages out between queues that have a binding to that exchange. It could quickly become problematic to do this manually, without adding too much information about numbers of queues and their bindings into the publisher.

The consistent hash exchange plugin can be used if you need to get maximum use of many cores in your cluster. Note that it's important to consume from all queues. Read more about the consistent hash exchange plugin here: https://github.com/rabbitmq/rabbitmq-consistent-hash-exchange.

RabbitMQ sharding

The RabbitMQ sharding plugin does the partitioning of queues automatically for you. Once you have defined an exchange as sharded, the supporting queues are automatically created on every cluster node and messages are sharded across them. RabbitMQ sharding shows one queue to the consumer, but it could be many queues running behind it in the background. The RabbitMQ Sharding plugin gives you a centralized place to where you can send your messages, plus load balancing, by adding queues to the other nodes in the cluster. Read more about RabbitMQ Sharding here: https://github.com/rabbitmq/rabbitmq-sharding.

Don’t set your own names on temporary queues

You can set a queue name or tell the broker to generate a name for your queues. Giving a queue a name is important when you want to share the queue between producers and consumers, but it's not important if you are using temporary queues. Instead, it would be best if you let the server choose a random queue name instead of making up your own names - or modify the RabbitMQ policies.

A queue name starting with amq. is reserved for internal use by the broker.

Auto-delete unused queues

Client connections can fail and potentially leave unused resources (queues) behind, and leaving too many queues behind might affect performance. There are three ways to have queues deleted automatically.

You can set a TTL policy on the queue. E.g., a TTL policy of 28 days deletes queues that haven't been consumed from in the last 28 days.

An auto-delete queue is deleted when its last consumer has canceled or when the channel/connection is closed (or when it has lost the TCP connection with the server).

An exclusive queue can only be used (consumed from, purged, deleted, etc) by its declaring connection. Exclusive queues are deleted when their declaring connection is closed or gone (e.g., due to underlying TCP connection loss).

Set limited use on priority queues

Queues can have 0 or more priorities. Each priority level uses an internal queue on the Erlang VM, which takes up some resources. In most use cases it's sufficient to have no more than 5 priority levels.

Payload - RabbitMQ message size and types

A common question is how to handle the payload size of messages sent to RabbitMQ. You should, of course, try to avoid sending very large files in messages, but messages per second is a way larger bottleneck than the message size itself. Sending multiple small messages might be a bad alternative. A better idea could be to bundle them into one larger message and let the consumer split it up. However, if you bundle multiple messages you need to keep in mind that this might affect the processing time. If one of the bundled messages fails - do you need to re-process them all? How you should set this up depends on bandwidth and your architecture.

Note: Don’t use too many connections or channels

Try to keep connection/channel count low.

Connections and channels

Each connection uses about 100 KB of RAM (and even more, if TLS is used). Thousands of connections can be a heavy burden on a RabbitMQ server. In the worst case, the server can crash due to OOM; it is running out of memory.

Try to keep connection/channel count low.

Don’t share channels between threads

Make sure that you don’t share channels between threads as most clients don’t make channels thread-safe (it would have a serious negative effect on the performance).

Note: Don’t share channels between threads

You should make sure that you don’t share channels between threads as most clients don’t make channels thread-safe.

Don’t open and close connections or channels repeatedly

Make sure that you don’t open and close connections or channels repeatedly - doing that gives you a higher latency, as more TCP packages have to be sent and received.

The handshake process for an AMQP connection is actually quite involved and requires at least 7 TCP packets (more if TLS is used). The AMQP protocol has a mechanism called channels that “multiplexes” a single TCP connection. It’s recommended that each process only creates one TCP connection, and uses multiple channels in that connection for different threads. Connections should also be long-lived, and channels can be opened and closed more frequently if required. Even channels should try to be long-lived if possible. Don’t open a channel every time you are publishing. Best practice is to reuse connections and multiplex a connection between threads with channels.

Note: Don’t open and close connections or channels repeatedly

Make sure that you don’t open and close connections or channels repeatedly - doing that gives you a higher latency, as more TCP packages have to be sent and received.

- AMQP connections: 7 TCP packages
- AMQP channel: 2 TCP packages
- AMQP publish: 1 TCP package (more for larger messages)
- AMQP close channel: 2 TCP packages
- AMQP close connection: 2 TCP packages

▸ Total 14-19 packages (+ Acks)

Separate connections for publisher and consumer

First of all; you might not be able to consume messages if you are using the same connection for publisher and consumer if the connection is in flow control, which will worsen the flow problem.

Secondly; RabbitMQ can apply back pressure on the TCP connection when the publisher is sending too many messages to the server. If you consume on the same TCP connection the server might not receive the message acknowledgments from the client. Thus, the consume performance is affected too. And with lower consume speed the server will be overwhelmed.

Note: Separate connections for publisher and consumer

Make sure to have separate connections for publisher and consumer to be able to get the highest throughput.

A large number of connections and channels might affect the RabbitMQ management interface performance

Another effect of having a large number of connections and channels is that the performance of the RabbitMQ management interface will slow down. Metrics have to be collected, analyzed and displayed for every connection and channel.

Acknowledgments and Confirms

Pay attention to where in your consumer logic you're acknowledging messages. Messages in transit might get lost in an event of a connection failure, and such a message might need to be retransmitted. Acknowledgments let the server and clients know when to retransmit messages. The client can either ack the message when it receives it, or when the client has completely processed the message. Acknowledgment has a performance impact, so for the fastest possible throughput, manual acks should be disabled.

A consuming application that receives essential messages should not acknowledge messages until it has finished whatever it needs to do with them so that unprocessed messages (worker crashes, exceptions, etc.) don't go missing.

Publish confirm, is the same thing, but for publishing. The server acks when it has received a message from a publisher. Publish confirm also has a performance impact. However, one should keep in mind that it's required if the publisher needs messages to be processed at least once.

Unacknowledged messages

All unacknowledged messages have to reside in RAM on the servers. If you have too many unacknowledged messages you will run out of memory. An efficient way to limit unacknowledged messages is to limit how many messages your clients prefetch. You can read more about this in the prefetch section.

Persistent messages and durable queues

You need to be prepared for broker restarts, broker hardware failure, or broker crashes. To ensure that messages and broker definitions survive restarts, we need to ensure that they are on disk. Messages, exchanges, and queues that are not durable and persistent are lost during a broker restart.

Make sure that your queue is declared as "durable" and your messages are sent with delivery mode "persistent".

Persistent messages are heavier as they have to be written to disk. Keep in mind that lazy queues have the same effect on performance, even though you are sending transit messages. For high performance - use transit messages. For high throughput use temporary, or non-durable queues.

Note: Use persistent messages and durable queues

If you cannot afford to lose any messages, make sure that your queue is declared as "durable", and your messages are sent with delivery mode "persistent" (delivery_mode=2).

TLS and AMQPS

You can connect to RabbitMQ over AMQPS, which is the AMQP protocol wrapped in TLS. TLS has a performance impact since all traffic has to be encrypted and decrypted. For maximum performance we recommend using VPC peering instead, then the traffic is being encrypted without involving the AMQP client/server.

At CloudAMQP we configure the RabbitMQ servers to only accept and prioritize fast but secure encryption ciphers.

Prefetch

The prefetch value is used to specify how many messages that are consumed at the same time. It's used to get as much out of your consumers as possible.

From RabbitMQ.com: “The goal is to keep the consumers saturated with work, but to minimize the client's buffer size so that more messages stay in RabbitMQ's queue and are thus available for new consumers or just to be sent out to consumers as they become free.”

RabbitMQ default prefetch setting gives clients an unlimited buffer, meaning that RabbitMQ by default send as many messages as it can to any consumer that looks ready to accept them. Messages that are sent are cached by the RabbitMQ client library (in the consumer) until it has been processed. Prefetch limits how many messages the client can receive before acknowledging a message. All pre-fetched messages are removed from the queue, and invisible to other consumers.

A too small prefetch count may hurt performance since RabbitMQ is most of the time waiting to get permission to send more messages. *Figure 37* is illustrating long idling time. In the example, we have a QoS prefetch setting of 1. This means that RabbitMQ won't send out the next message until after the round trip completes (deliver, process, acknowledge). Round time in this picture is in total 125ms with a processing time of only 5ms.

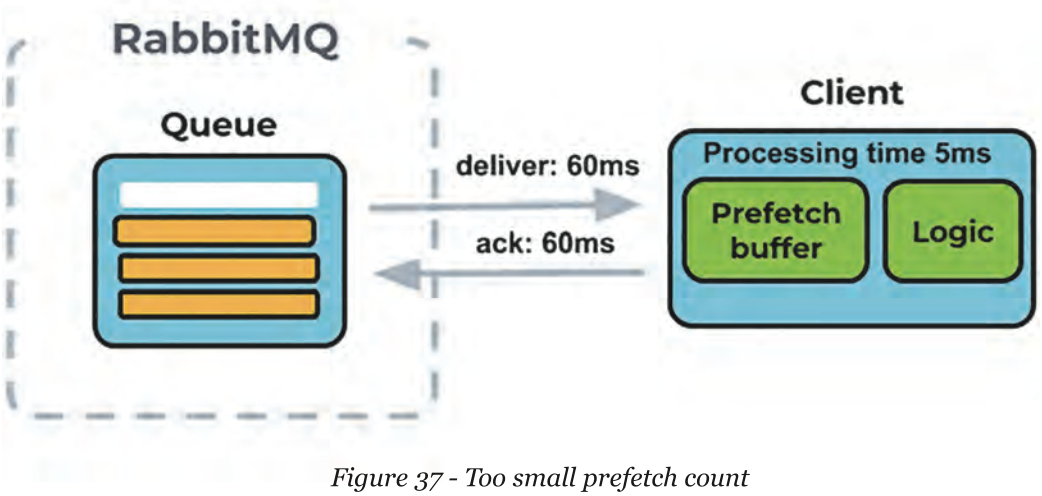


Figure 37 - Too small prefetch count may hurt performance in RabbitMQ.

A large prefetch count, on the other hand, could deliver lots of messages to one single consumer, and keep other consumers in an idling state.

How to set correct prefetch value?

If you have one single, or a few consumers, processing messages quickly, we recommend prefetching many messages at once. Try to keep your client as busy as possible. If you have about the same processing time all the time and network behavior remains the same - you can use the total round trip time/processing time on the client for each message, to get an estimated prefetch value.

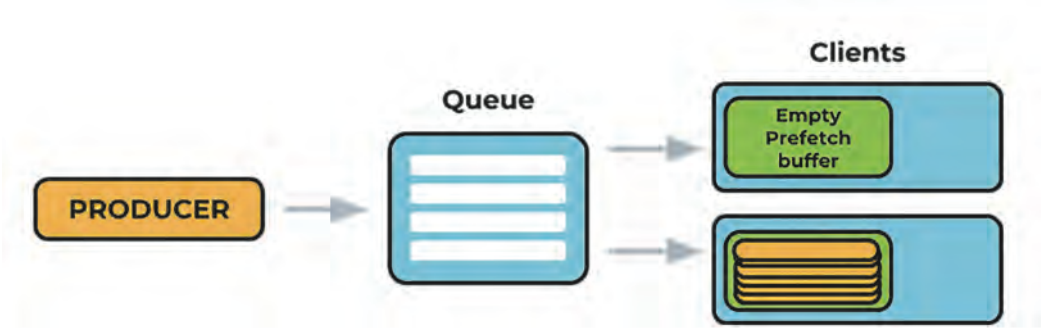


Figure 38 - A large prefetch count delivers lots of messages to one single consumer.

If you have many consumers, and short processing times, we recommend a lower prefetch value than for one single or few consumers. A too low value will keep the consumers idling a lot since they need to wait for messages to arrive. A too high value may keep one consumer busy, while other consumers are held in an idling state.

If you have many consumers, and/or long processing time, we recommend you to set prefetch count to 1 so that messages are evenly distributed among all your workers.

Please note that if your client auto-ack messages, the prefetch value have no effect.

Note: The prefetch value will have no effect if the client auto-ack messages.

A typical mistake is to have an unlimited prefetch, where one client receives all messages and runs out of memory and crashes, and then all messages are re-delivered again.

Note: Don't use an unlimited prefetch value

One client could receive all messages and then run out of memory.

HiPE

Enabling HiPE increases server throughput at the cost of increased startup time. When you enable HiPE, RabbitMQ is compiled at startup. The throughput increases with 20-80% according to our benchmark tests. The drawback of HiPE is that the startup time increases quite a lot too, about 1-3 minutes. HiPE is still marked as experimental in RabbitMQ's documentation.

Number of Nodes in your cluster (Clustering and High Availability)

When you create a CloudAMQP instance with one node, you get one single node with high performance. One node gives you the highest performance since messages are not mirrored between multiple nodes.

When you create a CloudAMQP instance with two nodes, you get half the performance compared to the same plan size for a single node. The nodes are located in different availability zones and queues are automatically mirrored between availability zones. Two nodes give you high availability since one node might crash or be marked as impaired, but the other node will still be up and running, ready to receive messages.

When you create a CloudAMQP instance with three nodes, you will get 1/4 of the performance compared to the same plan size for a single node. The nodes are located in different availability zones and queues are automatically mirrored between availability zones. You also get pause minority which means that by shutting down the minority component you reduce duplicate deliveries as compared to allowing every node to respond. Pause minority is a partition handling strategy in a three node cluster that protects data from being inconsistent due to net-splits.

Remember to enable HA on new vhosts

A common mistake we have noticed on CloudAMQP clusters is that users create a new vhost but forgets to enable an HA-policy for a new vhost. Messages will therefore not be synced between nodes.

Routing (exchanges setup)

Direct exchanges are the fastest to use. If you have many bindings, RabbitMQ has to calculate where to send the message.

Plugins

Some plugins might be super nice to have, but they might consume a lot of CPU or RAM usage. Therefore, they are not recommended for a production server. Make sure to disable plugins that you are not using. You are able to enable many different plugins via the control panel in CloudAMQP.

Note: Disable plugins that you are not using.

RabbitMQ management statistics rate mode

The RabbitMQ management interface collects and stores stats for all queues, connections, channels, etc. which might affect the broker in a negative way if you, for example, have too many queues. Setting the RabbitMQ management statistics rate mode to detailed could have a serious performance impact and should not be used in production.

Note: Don't set management statistics rate mode to detailed.

RabbitMQ and Erlang version and client libraries

Try to stay up-to-date with the latest stable versions of RabbitMQ and Erlang. CloudAMQP usually tests new major versions to a great extent before we release them for our customers. Please be aware that we always have the most recommended version as the selected option (default) in the dropdown of where you select a version for a new cluster.

Note: Don't use old RabbitMQ versions or RabbitMQ clients

Stay up-to-date with the latest stable versions of RabbitMQ and Erlang. Make sure that you are using the latest recommended version of client libraries.

Make sure that you are using the latest recommended version of client libraries. Check our documentation, and feel free to ask us if you have any questions regarding which library to use.

Use TTL with caution

Dead lettering and TTL are two popular features in RabbitMQ that should be used with caution. TTL and dead lettering can generate performance effects that you might not have foreseen.

Dead lettering

A queue that is declared with the x-dead-letter-exchange property sends messages which are either rejected, nacked or expired (with TTL) to the specified dead-letter-exchange. If you specify x-dead-letter-routing-key the routing key of the message will be changed when dead lettered.

TTL

By declaring a queue with the x-message-ttl property, messages will be discarded from the queue if they haven't been consumed within the time specified.

Best Practices For High Performance

It's time to maximize the message passing throughput in RabbitMQ. This section is a summary of recommended configurations for high-performance.

☑ Make sure your queues stay short

To get optimal performance, make sure your queues stay as short as possible all the time. Longer queues impose more processing overhead. We recommend that queues should always stay around 0 for optimal performance.

☑ Set a queue max-length if needed

A feature that could be recommended for applications that often get hit by spikes of messages, is to set a max-length on the queue. This keeps the queue short by discarding messages from the head of the queues so that it's never larger than the max-length setting.

☑ Do not use lazy queues

With lazy queues are messages automatically stored to disk, which will slow down the throughput. Please note that CloudAMQP has lazy queues enabled by default.

☑ Use transit messages

Persistent messages are written to disk as soon as they reach the queue, which lowers your throughput. Use transit messages to avoid this.

☑ Use multiple queues and consumers

Queues are single-threaded in RabbitMQ, and one queue can handle up to about 50k messages/s. You achieve better throughput on a multi-core system if you have multiple queues and consumers.

The RabbitMQ management interface collects and calculates metrics for every queue in the cluster. This might slow down the server if you have thousands upon thousands of active queues and consumers.

☑ Split your queues over different cores

Queue performance is limited to one CPU core. You will, therefore, get better performance if you split your queues into different cores, and also into different nodes if possible.

We recommend two plugins that will help you if you have multiple nodes or a single node cluster with multiple cores. Consistent hash exchange plugin and RabbitMQ sharding, as described in the previous chapter.

☑ Disable manual acks and publish confirms

Acknowledgment and publish confirms has a performance impact. For the fastest possible throughput, manual acks should be disabled. This will speed up the broker by allowing it to "fire and forget" the message.

☑ Avoid multiple nodes (HA)

One node gives you the highest throughput when compared to an HA cluster setup. Messages and queues are not mirrored to other nodes.

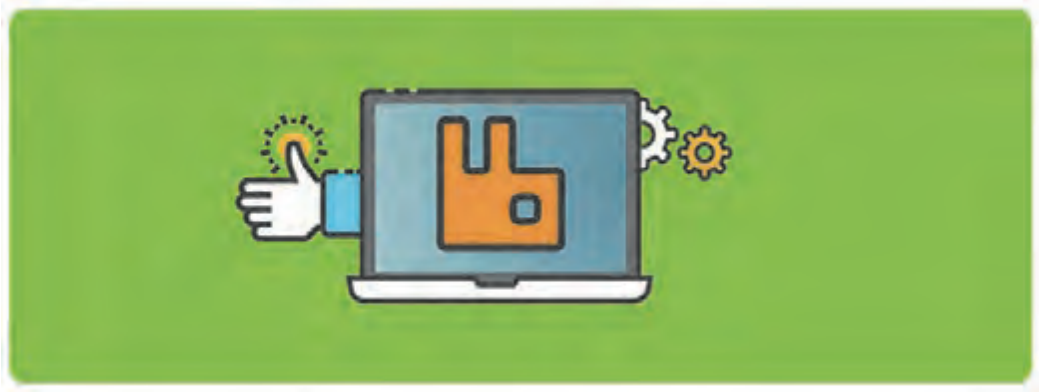
☑ Enable RabbitMQ HiPE

HiPE increase server throughput at the cost of increased startup time. When you enable HiPE, RabbitMQ is compiled at startup. The throughput increases with 20-80% according to our benchmark tests.

☑ Disable plugins you are not using

Some plugins might be super nice to have, but on the other hand, they might consume a lot. Therefore, they are not recommended for a production server. Make sure to disable plugins that you are not using.

Best Practices for High Availability



It's time to maximize the up-time for your RabbitMQ cluster. This section is a summary of recommended configurations for high-availability.

☑ Make sure your queues stay short

To get optimal performance, make sure your queues stay as short as possible all the time. Longer queues impose more processing overhead. We recommend that queues should always stay around 0 for optimal performance.

☑ Enable lazy queues

Lazy queues write messages to disk immediately, thus spreading the work out over time, instead of taking the risk of a performance hit somewhere down the road. This gives you a more predictable and smooth performance curve without any sudden drops.

☑ RabbitMQ HA - 2 nodes

Availability can be enhanced if clients can find a replica of data, even in the presence of failures. The ability to access the cluster even if a node in the cluster goes down. We recommend 2 nodes for high availability. CloudAMQP has located each node in a cluster in different availability zones (AWS), and queues are automatically mirrored, replicated (HA) between availability zones. Message queues are by default located on one single node but they are visible and reachable from all nodes.

When a node fails, we have a mechanism to auto-failover to other nodes in the cluster. We have added a load balancer in front of the RabbitMQ instances, which makes brokers distribution transparent from the message publishers. Maximum failover time in CloudAMQP is 60s (the endpoint health is measured every 30s, and the DNS TTL is set to 30s).

☑ Option to use federation between clouds

We do not recommend clustering between clouds or regions, and therefore no plan at CloudAMQP spread nodes across regions or datacenters. If one cloud region goes down, the CloudAMQP cluster also goes down - but don't be afraid, it's something that we have never experienced. Instead, cluster nodes are spread across availability zones within the same region.

You can protect the setup against a region-wide outage by setting up two clusters in different regions and use federation between them. Federation is one of the ways by which a software system can benefit from having multiple RabbitMQ brokers distributed on different machines.

☑ Send persistent messages to durable queues

In order to avoid losing messages in the broker, you need to be prepared for broker restarts, broker hardware failure, and broker crashes. To ensure that messages and broker definitions survive restarts, we need to ensure that they are on disk. Messages, exchanges, and queues that are not durable and persistent are lost during a broker restart.

Make sure that your queue is declared as “durable”, and your messages are sent with delivery mode "persistent".

☑ Do not enable HiPE

HiPE will increase server throughput at the cost of increased startup time. When you enable HiPE, RabbitMQ is compiled at startup. The drawback is that the startup time increases quite a lot too, 1-3 minutes. HiPE might affect uptime during a server restart, which affects availability.

☑ Management statistics rate mode in production

Setting RabbitMQ management statistics rate mode to detailed has a serious performance impact and should not be used in production.

☑ Limited use of priority queues

Each priority level uses an internal queue on the Erlang VM, which takes up some resources. In most use cases it's sufficient to have no more than 5 priority levels.

Queue Federation

RabbitMQ supports federated queues which have several use-cases. When collecting messages from multiple clusters to a central cluster. When distributing the load of one queue to multiple other clusters or when migrating to another cluster without stopping all producers/consumers while doing so.

Queue federation connects an upstream queue to transfer messages to the downstream queue when there are consumers that have capacity on the downstream queue. It's perfect to use when migrating between two clusters. Consumers and publishers can be moved in any order, and the messages won't be duplicated (which is the case if you do exchange federation). Instead, messages are transferred to the new cluster when your consumers are there. The federated queue will only retrieve messages when it has run out of messages locally, it has consumers that need messages, and the upstream queue has "spare" messages that are not being consumed.

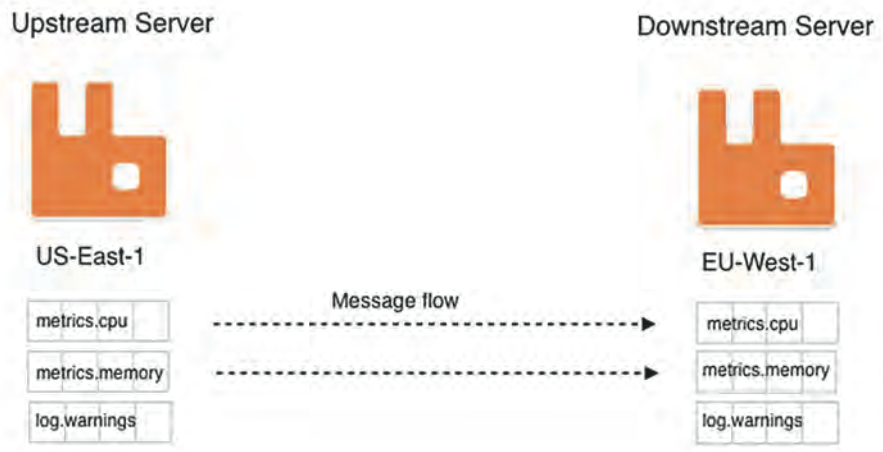


Figure 39 - Upstream and downstream servers.

Let us start by defining the concept of **upstream** and **downstream** servers *(figure 39)*. The upstream server is where messages are initially published. Downstream servers are where the messages get forwarded to. The downstream cluster can be thought of as subscribing to messages from the upstream cluster.

Queue Federation example setup

In this example, there is already one cluster set up in Amazon US-East-1 (named: dupdjffe, as seen in *figure 40*). This cluster is going to be migrated to a cluster in EU-West-1 (in the pictures named: aidajcdt). In this case, the server in US-East-1 will be defined as the upstream servers of the servers in EU-West-1.

Some queues in the cluster in US-East-1 are going to be migrated via federation, the metric-queues (metric.cpu and metric.memory).

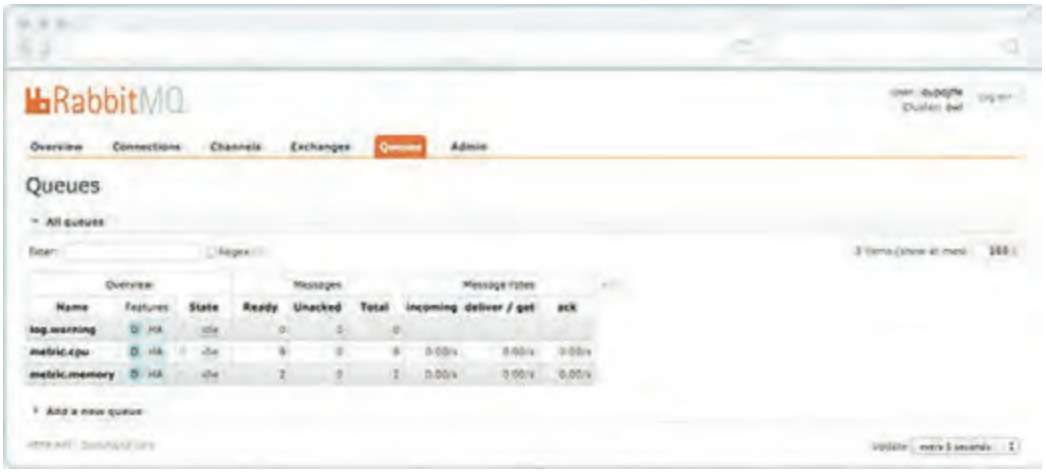


Figure 40 - Migration of metric.cpu and metric.memory.

1. Set up the new cluster

Start by setting up the new cluster. In this case the cluster in EU-West-1.

2. Create a policy that matches the queues that you would like to federate.

A policy is a pattern that queue names are matched against. The "pattern" argument is a regular expression used to match queue (or exchange) names. In this case, we tell the policy to federate all queues whose name begin with "metric.".

Navigate to *Admin* -> *Policies* and press *Add/update* a policy to create the policy *(figure 41)*. A policy can apply to an upstream set or a single upstream of exchanges and/or queues. In this example it's applied to all upstreams, federation-upstream-set is set to all.

NOTE: Policies are matched every time an exchange or queue is created.

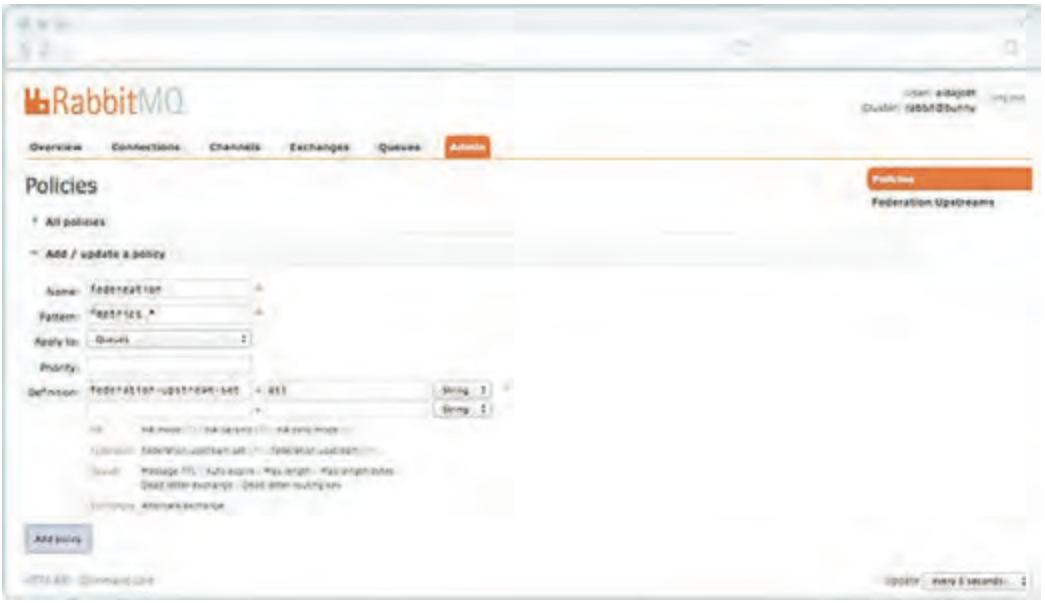


Figure 41 - Apply federation policy.

3. ***Start by setting up the new cluster. In this case the cluster in EU-West-1.***

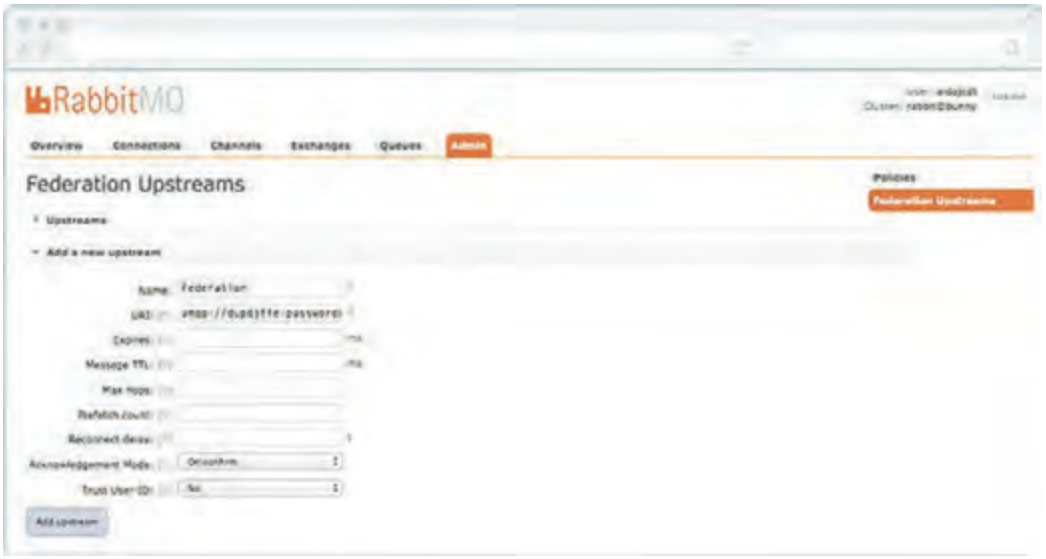


Figure 42 - Set up the federation to the upstream

Open the management interface for the downstream server (EU-West-1) and go to the *Admin -> Federation Upstreams* screen and press Add a new upstream *(figure 42)*. Fill in all information needed, the URI should be the URI of the upstream server.

Leave expiry time and TTL blank. Leaving this blank means that it will stay forever.

4. **Start to move messages**

It's time to connect or move the publisher or the consumer to the new cluster. If you are migrating the cluster, you can move the publisher and/or the consumer in any order. The federated queue will only retrieve messages when it has run out of messages locally when it has consumers that need messages, or when the upstream queue has "spare" messages that are not being consumed.

5. **Verify that messages are federated**

Verify that the downstream server consumes the messages published to the queue at the upstream server when there are consumers available at the downstream server. If it does, then we are done.

RabbitMQ Protocols

RabbitMQ is an open source multi-protocol messaging broker. This means that RabbitMQ supports several messaging protocols, over a range of different open and standardized protocols such as AMQP, HTTP, STOMP, MQTT and WebSockets/Web-Stomp.

Most message queuing protocols have many features in common; however, some are better suited to a particular set of use cases than others. In the simplest case, a message queue is using an asynchronous protocol in which the sender and the receiver do not operate on the message at the same time.

The protocol only defines the communication between the client and the server and has no impact on a message itself. One protocol can be used when publishing and you can consume using another protocol. MQTT with its minimal design makes it perfect for built-in systems, mobile phones, and other memory and bandwidth sensitive applications. A specific task may as well be achieved using AMQP, but MQTT might be a more appropriate choice of protocol for this specific type of scenarios.

When you create a CloudAMQP instance are all the common protocols available by default (the web-stomp plugin/WebSockets is only enabled on dedicated plans).

AMQP

RabbitMQ was originally developed to support AMQP which is the "core" protocol supported by the RabbitMQ broker. AMQP stands for Advanced Message Queuing Protocol and it's an open standard application layer protocol. RabbitMQ implements version 0-9-1 of the specification today, with legacy support for version 0-8 and 0-9. AMQP was designed to efficiently support a wide variety of messaging applications and communication patterns. AMQP is a more advanced protocol than MQTT, more reliable and has better support for security. AMQP also have features like flexible routing, durable and persistent queues, clustering, federation, and high availability queues. The downside is that it's a more verbose protocol on the wire - depending on how you implement your solution.

As with other message queuing protocols, the defining features of AMQP are message orientation and queuing. Routing is another feature, the process by which an exchange decides which queues

to place your message on. Messages in RabbitMQ are routed from the exchange to the queue depending on exchange types and keys. Reliability and Security are other important features of AMQP. RabbitMQ can be configured to ensure that messages are always delivered, read more in the Reliability Guide at www.rabbitmq.com/reliability.html.

For more information about AMQP, check out the AMQP Working Group's overview page.

CloudAMQP AMQP assigned port number is 5672 or 5671 for AMQPS (TLS/SSL encrypted AMQP).

MQTT

MQ Telemetry Transport is a publish-subscribe pattern-based "lightweight" messaging protocol. The protocol is often used in the IoT (Internet of Things) world of connected devices. Its designed for built-in systems, mobile phones, and other memory and bandwidth sensitive applications.

MQTT advertised benefits for IoT only matter for extremely low power devices. MQTT is very wire-efficient, it has a strong focus on minimal wire footprint. It requires less effort to implement MQTT on a client than AMQP - because of its simplicity. However, MQTT lacks authorization and error notifications from the server to clients, which might be considered as significant limitations.

CloudAMQP MQTT assigned port number is 1883 (8883 for TLS wrapped MQTT). Use the same default username and password as for AMQP.

STOMP

STOMP, Simple (or Streaming) Text Oriented Message Protocol, is a simple text-based protocol used for transmitting data across applications. It's a much simpler and less complex protocol than AMQP, with more similarities to HTTP. STOMP clients can communicate with almost every available STOMP message broker, this provides easy and widespread messaging interoperability among many languages, platforms, and brokers. It's for example possible to connect to a STOMP broker using a telnet client.

STOMP does not deal with queues and topics — it uses a SEND semantic with a destination string. RabbitMQ maps the message to topics, queues or exchanges (other brokers might map onto something else that it understands internally). Consumers then SUBSCRIBE to those destinations.

STOMP is recommended if you are implementing a simple message queuing application without complex demands on a combination of exchanges and queues. RabbitMQ supports STOMP (all current versions) via the STOMP plugin.

CloudAMQP STOMP assigned port number is 1883, 61613 (61614 for TLS wrapped STOMP).

HTTP

HTTP stands for Hypertext Transfer Protocol and it's an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP is not a messaging protocol. However, RabbitMQ can transmit messages over HTTP.

The rabbitmq-management plugin provides an HTTP-based API for management and monitoring of your RabbitMQ server.

CloudAMQP HTTP assigned port number is 443.

Publish with HTTP

Example of how to publish a message to the default exchange with the routing key "my_key":

```
curl -XPOST -d'{"properties":{},"routing_key":"my_key","payload":"my body","p
ayload_encoding":"string"}' https://username:password@hostname/api/exchanges/
vhost/amq.default/publish
Response: {"routed":true}
```

Get message with HTTP

Example of how to get one message from the queue "your_queue". (This is not an HTTP GET as it will alter the state of the queue.)

```
curl -XPOST -d'{"count":1,"requeue":true,"encoding":"auto"}' https://user:pas
s@host/api/queues/your_vhost/your_queue/get
Response: Json message with payload and properties.
```

Get queue information

```
curl -XGET https://user:pass@host/api/queues/your_vhost/your_queue
Response: Json message with queue information.
```

Autoscale by polling queue length

When jobs are arriving faster in the queue than they are processed - and when the queue starts growing in length, it's a good idea to spin up more workers. By polling the HTTP API queue length, you can spin up or take down workers depending on the length.

Web-Stomp

Web-Stomp is a plugin to RabbitMQ which exposes a WebSockets server (with fallback) so that web browsers can communicate with your RabbitMQ server/cluster directly.

To use Web-Stomp you first need to create at least one user, with limited permissions, or a

new vhost which you can expose publicly because the username/password must be included in your javascript, and a non-limited user can subscribe and publish to any queue or exchange.

The Web-Stomp plugin is only enabled on dedicated plans on CloudAMQP.

Next include `socks.min.js` and `stomp.min.js` in your HTML from for example CDNJS:

```
  <script src="//cdnjs.cloudflare.com/ajax/libs/sockjs-client/0.3.4/sockjs.mi
n.js"<</script>
  <script src="//cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.min.js">
</script>
```

To connect:

```
// Replace with your hostname
var host = "https://host.rmq.cloudamqp.com/stomp";
var ws = new SockJS(host);
var client = Stomp.over(ws);

// RabbitMQ SockJS does not support heartbeats
client.heartbeat.outgoing = 0;
client.heartbeat.incoming = 0;

client.debug = onDebug;

// Make sure the user has limited access rights
client.connect("ws-user", "ws-password",
  onConnect, onError, "vhost");

function onConnect() {
  var id = client.subscribe("/exchange/web/chat",
    function(d) {
      var node = document.createTextNode(d.body + '\n');
      document.getElementById('chat').appendChild(node);
    }
  );
}

function sendMsg() {
  var msg = document.getElementById('msg').value;
  client.send('/exchange/web/chat', {
    "content-type": "text/plain"
```

```
  }, msg);
}

function onError(e) {
  console.log("STOMP ERROR", e);
}

function onDebug(m) {
  console.log("STOMP DEBUG", m);
}
```

Part 3: RabbitMQ User Stories

A small story about how someone uses RabbitMQ can always be of great value. Here follow some user stories from our customers, and from us, that describes how RabbitMQ is used in production.

A growing digital parking service from Sweden is right now breaking down their monolithic application towards microservices. Follow their story! (This story was written in September 2017.)

A portable parking meter in your pocket

Founded in 2010, Parkster has quickly become one of the fastest growing digital parking services in Sweden. Their vision is to make it quick and easy for you to pay your parking fees with your smartphone, via your Parkster app, SMS or voice mail. They want to see a world where you don't need to guesstimate the required parking time or stand in line waiting by a busy parking meter. It should be easy to pay for parking - for everyone, everywhere. Moreover, Parkster doesn't want the customer to pay more when using tools of the future - that's why there are no extra fees when you are using Parkster's app for parking.

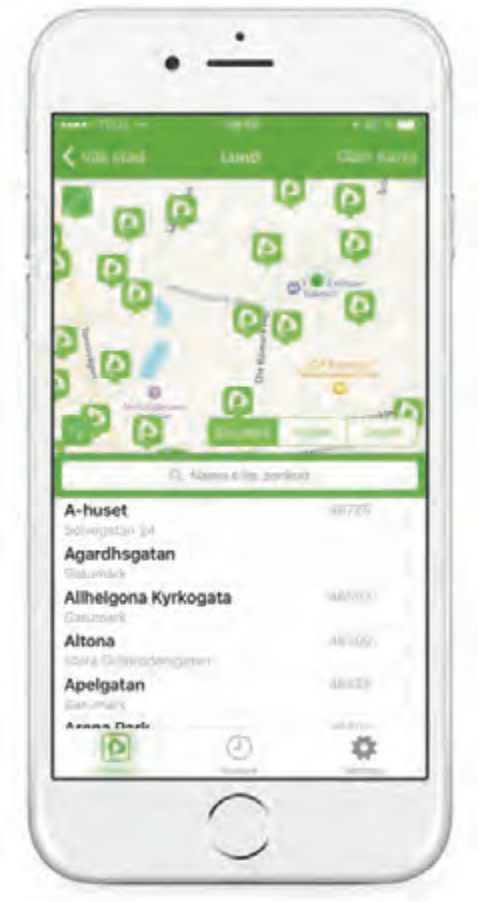


Figure 43 - Parkster iPhone App

Breaking up a tightly coupled monolithic application

Like many other companies, Netflix just to mention one, Parkster started out with a monolithic architecture. They wanted to have their business model proven before they went further. A monolithic application is where the whole application is built as a single unit. All code for a system is in a single codebase that is compiled together and produces a single system.

Having one codebase seemed like the easiest and fastest solution at the time, and solved their core business problems, which included connecting devices with people, parking zones, billing, and payments. A few years later, they decided to break up the monolith into multiple small codebases, which they did through multiple microservices communicating via message queues.

Parkster tried out their parking service for the first time in Lund, Sweden. After that, they rapidly expanded into more cities and introduced new features. The core model grew and components became tightly coupled.

Deploying the codebase meant deploying everything at once. One big codebase made it hard and difficult to fix bugs and to add new features. A deep knowledge was also required before doing an attempt for a single small code change, no one wants to add new code that could disrupt operation in some unforeseen way. One day they had enough - the application had to be decoupled.

Application decoupling

Parkster's move from a monolith architecture to a microservice architecture is right now a work in progress. They are breaking up their software into a collection of small, isolated services, where each service can be deployed and scale as needed - independently of other services. Their system today has about 15-20 microservices, where the core app is written in Java.

They are already enjoying their changes: "It's very nice to focus on a specific limited part of the system instead of having to think about the entire system, every time you do something new or make changes. As we grow, I think we will benefit even more from this change," said Anders Davoust, a developer at Parkster.

Breaking down their codebase has also given the software developers the freedom to use whatever technologies that make sense for a particular service. Different parts of the application can evolve independently, be written in different languages and/or maintained by separated developer teams. For example, one part of the system uses MongoDB and another part uses MySQL. Most code is written in Java, but parts of the system are written in Clojure. Parkster is using the open-source system Kubernetes as a container orchestration platform.

Resiliency - The capacity to recover quickly

Applications might be delayed or crash sometimes - it happens. It could be due to timeouts or that you simply have errors in your code that could affect the whole application.

Another thing Parkster really like about their system today is that it can still be operational even if part of the backend processing is delayed or broken. Everything will not break just because one small part of the system is not operating as it should. By breaking up the system into autonomous components, Parkster inherently becomes more resilient.

Message queues, RabbitMQ and CloudAMQP

Parkster is separating different components via message queues. A message queue may force the receiving application to confirm that it has completed a job and that it's safe to remove the job from the queue. The message will just stay in the queue if anything fails in the receiving application. A message queue provides a temporary message storage when the destination program is busy or not connected.

The message broker used between all microservices in Parkster is RabbitMQ. “It was a simple choice - we had used RabbitMQ in other projects before we built Parkster and we had a good experience with RabbitMQ.” The reason they went for CloudAMQP, was because they felt that CloudAMQP had more knowledge about the management of RabbitMQ than they had. They simply wanted to put their focus on the product instead of spending days configuring and handling server setups. CloudAMQP has been at the forefront when it comes to RabbitMQ server configurations and optimization since 2012.

I asked what they like about CloudAMQP, and I received a quick answer: “I love the support that CloudAMQP gives us, always quick feedback and good help”.

Now, Parkster’s goal is to get rid of the old monolithic repo entirely and focus on a new era where the whole system is built upon microservices.

Figure 44 - Parkster HQ in Lund, Sweden

A Microservice Architecture built upon RabbitMQ

We at CloudAMQP rely on RabbitMQ in our everyday life - a huge part of our events in the production environment are passing through RabbitMQ. This chapter gives a simple overview of the automated process behind CloudAMQP, the polyglot workplace where microservices written in different languages communicate through RabbitMQ.

CloudAMQP never had a traditional monolithic set up. It's built from scratch on small, independent, manageable services that communicate with each other. These services are all highly decoupled and focused on their task. This chapter gives an overview and a deeper insight into the automated process behind CloudAMQP. It describes some of our microservices and how we are using RabbitMQ as message broker when communicating between services.

Background of CloudAMQP

A few years ago Carl Hörberg (CEO @CloudAMQP) saw the need for a hosted RabbitMQ solution. At the time he was working at a consultancy company where he was using RabbitMQ in combination with Heroku and AppHarbor. He was looking for a hosted RabbitMQ solution himself, but he could not find any. shortly after, he started CloudAMQP, which entered the market in 2012.

The automated process behind CloudAMQP

CloudAMQP is built upon multiple small microservices, where RabbitMQ is used as a messaging system. RabbitMQ is responsible for the distribution of events to the services that listen for them. We have the option to send a message without having to know if another service is able to handle it immediately or not. Messages can wait until the responsible service is ready. A service publishing a message does not need to know anything about the inner workings of the services that process that message. We follow the pub-sub (publish-subscribe) pattern and we rely on retry upon failure.

When you create a CloudAMQP instance you get the option to choose a plan and how many nodes you would like to have. The cluster will behave a bit different depending on the cluster setup. You also get the option to create your instance in a dedicated VPC and select RabbitMQ version.

A dedicated RabbitMQ instance can be created via the CloudAMQP control panel, or by adding the CloudAMQP add-on from any of our integrated platforms, like Heroku, IBM Cloud Catalogue, AWS marketplace, Azure marketplace, Manifold just to mention a few.

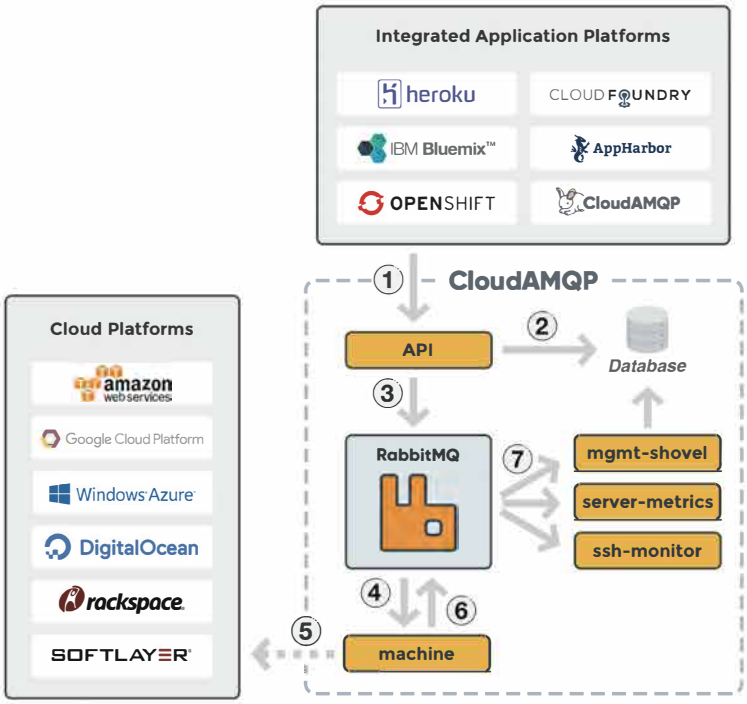


Figure 45 - The automated process behind CloudAMQP

When a client creates a new dedicated instance, an HTTP request is sent from the reseller to a service called CloudAMQP-API (1). The HTTP request includes all information specified by the client: plan, server name, data center, region, number of nodes etc., as you see in the picture above. CloudAMQP-API handles the request, save information into a database (2), and finally, send a account.create-message to one of our RabbitMQ-clusters (3).

Another service, called CloudAMQP-machine, is subscribing to account.create. CloudAMQP-machine takes the account.create-message of the queue and performs actions for the new account (4).

CloudAMQP-machine trigger a whole lot of scripts and processes. First, it creates the new server(s) in the chosen datacenter via an HTTP request (5). We use different underlying instance types depending on data center, plan and number of nodes. CloudAMQP-machine is responsible for all configuration of the server, setting up RabbitMQ, mirror nodes, handle clustering for RabbitMQ etc., all depending on the number of nodes and chosen datacenter.

CloudAMQP-machine sends a account.created-message back to the RabbitMQ cluster once the cluster is created and configured. The message is sent on the topic exchange (6). The great thing about the topic exchange is that a whole lot of services can subscribe to the event. We have a few

services that are listening to account.created-messages (7). Those services will all set up a connection to the new server. Here are three examples of services that are receiving the message and starts to work towards the new servers.

CloudAMQP-server-metrics: continuously gathering server metrics, such as CPU and disk space, from all servers.

CloudAMQP-mgmt-shovel: continuously ask the new cluster about RabbitMQ specific data, such as queue length, via the RabbitMQ management HTTP API.

CloudAMQP-SSH-monitor: is responsible for the monitoring of many processes that need to be running on all servers.

CloudAMQP does, of course, have a lot of other services communicating with the services described above and with the new server(s) created for the client.

CloudAMQP-server-metrics

As said before CloudAMQP-server-metrics is collecting metrics (CPU/memory/disk data) for ALL running servers. The collected metric data from a server is sent to RabbitMQ queues defined for the specific server e.g., server.<hostname>.vmstat and server.<hostname>.free where hostname is the name of the server.

CloudAMQP-alarm

We have different services that are subscribing to this data. One of these services is called CloudAMQP-alarm. A user is able to enable/disable CPU and memory alarms for an instance. CloudAMQP-alarm check the server metrics subscribed from RabbitMQ against the alarm thresholds for the given server, and it notifies the owner of the server if needed.

CloudAMQP-put-metrics

CloudAMQP has integrated monitoring tools; DataDog, Logentries, AWS Cloudwatch, Google Cloud Stackdriver Logging and Librato. A user who has an instance up and running have the option to enable these integrations. Our microservice CloudAMQP-put-metrics checks the server metrics subscribed from RabbitMQ and is responsible for sending metrics data to tools that the client has integrated with.

Figure 46 - The microservice CloudAMQP-put-metrics

This section described a small part of the CloudAMQP service. We do in total have around 100 microservices, all communicating via CloudAMQP and RabbitMQ.

I hope you enjoyed the reading and that you at least learned someting new. Never hesitate to send me an email!

TIMES ARE CHANGING!

Reliability and scalability are more important than ever. Therefore companies are rethinking their architecture. Monoliths are evolving into microservices and servers are moving into the cloud. Message Queues and especially RabbitMQ has come to play a significant role in the growing world of microservices. After all, it's one of the most widely deployed open source message brokers.

This book will help you along your RabbitMQ journey. It consists of three main parts:

- Introduction to Message Queuing and RabbitMQ
- Advanced Message Queuing with RabbitMQ
- User Stories

Essentially, this book is about RabbitMQ;
and who knows about queuing better than a bunch of Swedes*?

* In case you haven't heard: queuing is an integral part of the Swedish everyday life.

This book is provided by the Swedish tech company, 84codes AB, which provides the service CloudAMQP - RabbitMQ as a service.

ISBN 978-91-639-9585-9