SECOND EDITION

THE OPTIMAL RABBITMQ GUIDE

FROM BEGINNER TO ADVANCED

BY LOVISA JOHANSSON

OVER 26,000+
READERS & COUNTING

THE OPTIMAL RABBITMQ GUIDE

From Beginner to Advanced

I want to say a big thank you to everyone who has helped me, from the earliest draft of this book, up to this edition. A special thanks go out to my lovely colleagues at 84codes and my tech friends.

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!

- Lovisa Johansson, CloudAMQP

The Optimal RabbitMQ Guide

Book version: 2.1

Author: Lovisa Johansson

Email: lovisa@cloudamqp.com

Co-author: Elin Vinka

Email: elin@cloudamqp.com

Graphics: Daniel Marklund, Elin Vinka

Published: 2020-03-10

ISBN: 978-91-519-3115-9

I'd love to hear from you

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

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

PART ONE

PART TWO

PART THREE

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 in particular, as one of the most widely deployed open-source message brokers, have come to play a significant role in the growing world of microservices. This book is divided into three parts:

The first part is an introduction to microservices and RabbitMQ, including a section on how to use RabbitMQ on the CloudAMQP platform. This section includes the most important RabbitMQ concepts and how it allows users to create products that meet current and future industry demands.

The second part of the book is for more advanced users, who will learn how to take full advantage of RabbitMQ. This section explores Best Practice recommendations for High Performance and High Availability and common RabbitMQ errors and mistakes. It is also a deep dive into some RabbitMQ concepts and features.

The third part gives some real-world user stories from our own experiences with RabbitMQ as well from a CloudAMQP customer's point of view.

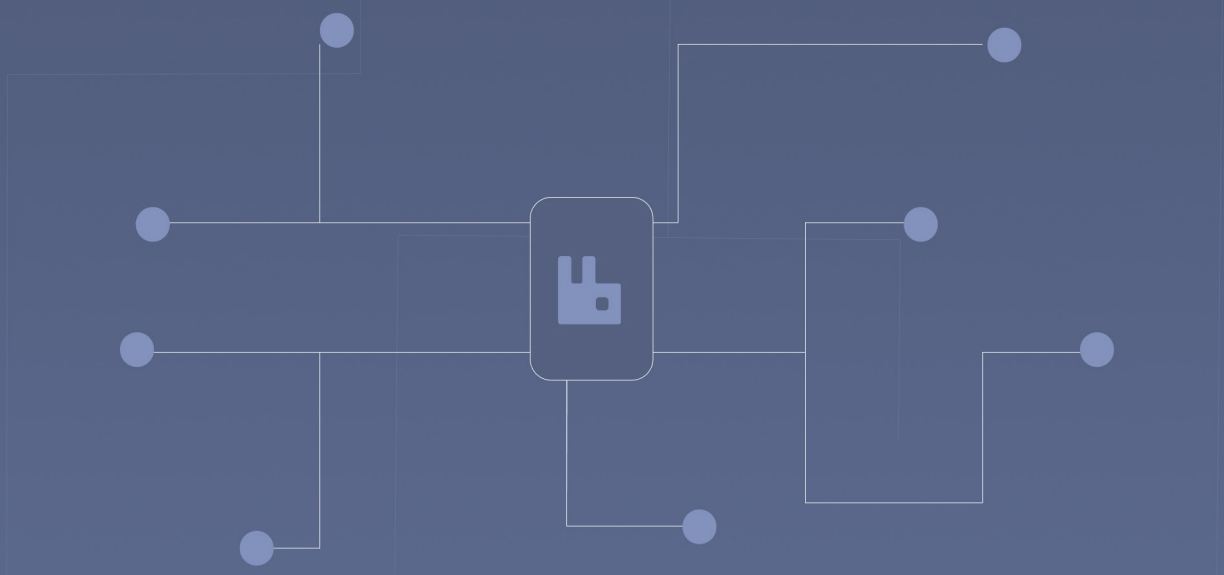
We hope that this book takes you far on your message queuing journey with RabbitMQ.

PART ONE

INTRODUCTION TO RABBITMQ

A MICROSERVICE ARCHITECTURE

Modules with various functionalities operate together to form a complete software application via properly articulated operations and interfaces.



Welcome to the wonderful world of message queuing! Many modern cloud architectures are a collection of loosely coupled microservices that communicate via a message queue. This microservice architecture is beneficial in that all the different parts of the system are easier to develop, deploy and maintain.

One of the advantages of a message queue is that it makes your data temporarily persistent, reducing the chance of errors that may occur when different parts of the 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 features basic information on microservices and message queuing, and the benefits of RabbitMQ.

PART ONE

MICROSERVICES AND RABBITMQ

Managing a complex, unified enterprise application can be a lot more difficult than it seems. Before making even a small change, hours of testing and analysis are required for possible impacts your change could have on the overall system. New developers must spend days getting familiar with the system before they are considered ready to write a non-breaking line of code. Fortunately, the situation becomes far less complicated thanks to the use of microservices. This chapter talks about the benefits of a microservice architecture.

In a microservice architecture, also called modular architecture, modules are decoupled from each other. These modules often offer various functionalities that operate together to form 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 adopting microservice architecture because it allows for 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 having to wait through years of testing.

BENEFITS OF A MICROSERVICE ARCHITECTURE

- **Easier development and maintenance**

  Imagine building a huge, bulky billing application that will involve authentication, authorization, charging and reporting of transactions. Dividing the application across multiple services (one for each functionality) will separate the responsibilities and 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, updating an authentication scheme will only require the addition of code to the authentication module and testing without having to worry about disrupting any other functionalities of the application.

- **Fault isolation**

  Another obvious advantage offered by a microservice architecture is its ability to isolate the fault to a single module. For example, if a reporting service is down, authentication and billing services will still be running, ensuring that customers can perform important transactions even when they are not able to view reports.

- **Increased speed and productivity**

  Microservice architecture is all about decoupling functions into manageable modules 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 each microservice can be tested independently to determine the readiness of the overall system.

- **Improved scalability**

  Microservices also allow effortless system scaling whenever required. Adding new components to just one service can be completed without changing any of the other services. Along the same lines, resource-intensive services can also be deployed across multiple servers by using microservices.

- **Easy to understand**

  Another advantage offered by a microservice architecture is the ease of understandability. Because each module represents a single functionality, getting to know the relevant details becomes easier and faster. For example, a consultant that works on charging services does not have to understand the whole system to perform maintenance or enhancements to their part of the system.

THE ROLE OF A MESSAGE QUEUE IN A MICROSERVICE ARCHITECTURE

Think of a message broker like a delivery person who takes mail from a sender and delivers it to the correct destination. In a microservice architecture, there typically are cross-dependencies that mean no single service can perform without getting help from other services. This is where it is crucial for systems to have a mechanism in place, allowing services to keep in touch with each other with no blocked responses. Message queuing fulfils this purpose by providing a means for services to push messages to a queue asynchronously and ensure they are delivered to the correct destination. To implement message queuing, a message broker is required.

PART ONE

WHAT IS RABBITMQ?

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

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

RabbitMQ is a message queuing software also known as a message broker or queue manager. Simply put, it is a software where queues can be defined, and applications may connect 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 or job that should start on another application, possibly even on another server, or it might be a simple text message.

The message broker stores the messages until a receiving application connects and consumes a message off the queue. The receiving application then appropriately processes the message. A message producer adds 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. A task that would generally take a lot of time to process can be delegated to a third party whose only job is to perform them.

This chapter is a case study of a web application, a PDF generator. The application allows users to upload information to a website. The website handles the information and generates a PDF, which is then emailed back to the user. The whole processing in this example takes several seconds to perform.

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 such as name and email. The message is then placed onto a queue defined in RabbitMQ using an exchange process. The underlying architecture of a message queue is simple - client applications called producers create messages and deliver them to the broker. The broker is, essentially, the message queue. Other applications, known as consumers, connect to the queue and subscribe to messages from the broker, processing those messages according to their function. The software interacting with the broker can be a producer, a consumer, or both. 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.

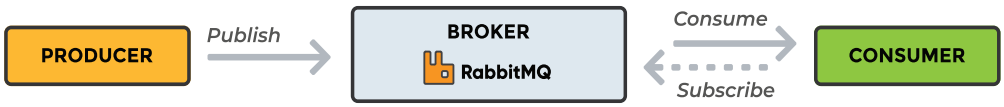


***Figure 2** - A sketch of the RabbitMQ workflow.*

WHY AND WHEN TO USE RABBITMQ

Message queuing allows web servers to respond to requests in their own time instead of being forced to perform resource-heavy procedures immediately. Message queuing is also useful for distributing a message to multiple recipients for consumption or when balancing the load between workers.

The consumer application removes a message from the queue and processes the PDF at the same time as the producer pushes new task 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, as the two applications only communicate through the messages they are sending to each other. The two services have what is known as 'low coupling' between the sender and the receiver.

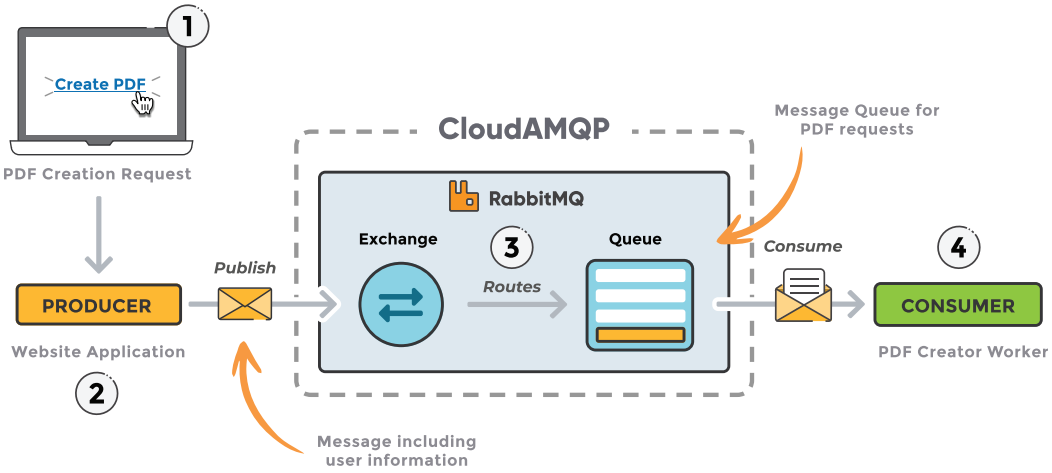


***Figure 3** - Application architecture example with RabbitMQ.*

Example

1. The user sends a PDF create request to the web application.
2. The web application (the producer) sends a message to RabbitMQ that includes 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 processor (the consumer) receives the job message from the queue 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. 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.

Message Flow in RabbitMQ

1. The producer publishes a message to an exchange. When creating an exchange, its type must be specified. The different types of exchanges are explained in detail later on in this book.

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 these attributes.

4. The messages stay in the queue until the consumer processes them.

5. The consumer then removes the message from the queue once handled.

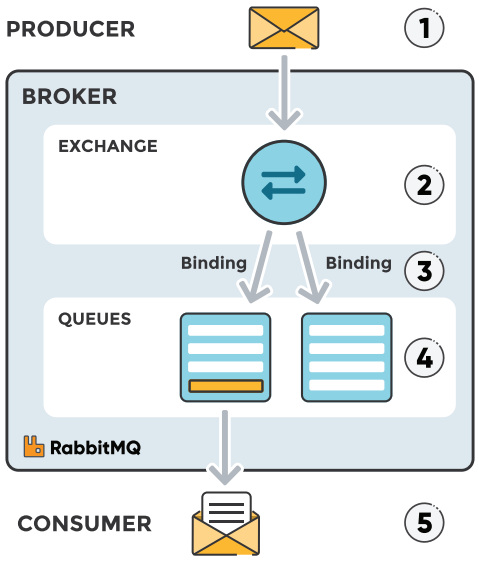


Figure 4 - Illustration of the message flow in RabbitMQ

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 with the exact match of binding key as the routing key of the message. For example, if the queue is bound to the exchange using 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** - A header exchange uses the message header attributes for routing purposes.

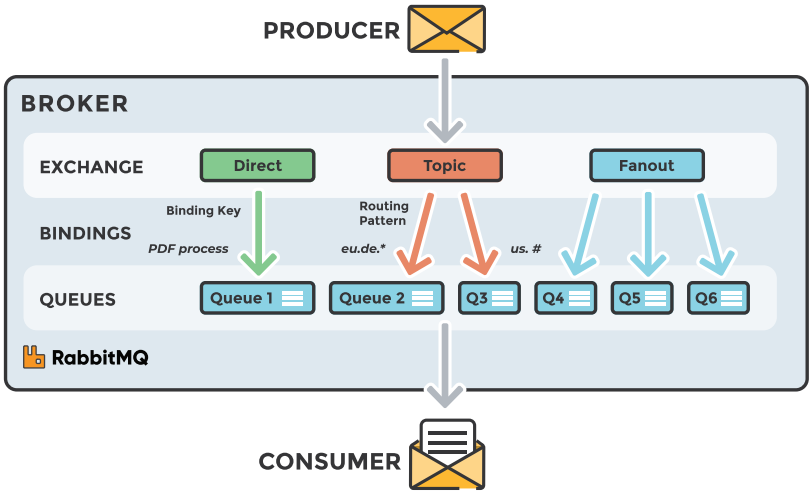


Figure 5 - Three different exchanges: direct, topic, and fanout.

RABBITMQ AND SERVER CONCEPTS

Below 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 sent from producer to consumer through RabbitMQ
- **Connection** - TCP connection between the application and the RabbitMQ broker
- **Channel** - A virtual connection inside a *connection*. When 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** - Link between a queue and an exchange
- **Routing Key** - The key that the exchange looks at to decide how to route the message to queues. Think of the routing key as the destination address of a message.
- **AMQP** - Advanced Message Queuing Protocol - the primary protocol used by RabbitMQ for messaging.
- **Users** - It's possible to connect to RabbitMQ with a given username and password, with assigned permissions such as rights to read, write and configure. Users can also be assigned permissions to specific virtual hosts.
- **Vhost** - Virtual host or Vhost 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** - Indicators that messages have been received or acted upon. Acknowledgements can be used in both directions; for example, 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, set up a CloudAMQP instance or set up RabbitMQ on a local machine. CloudAMQP is a hosted RabbitMQ solution (RabbitMQ as a Service), meaning that all that is required is to sign up for an account and create an instance. There is no need to set up and install RabbitMQ or care about cluster handling, as CloudAMQP will handle that. RabbitMQ is available free with the plan Little Lemur. Go to the plan page (www.cloudamqp.com/plans.html) and sign up for an appropriate plan. Click on "details" of the cloud-hosted RabbitMQ instance to find the username, password, and connection URL (Figure 7).

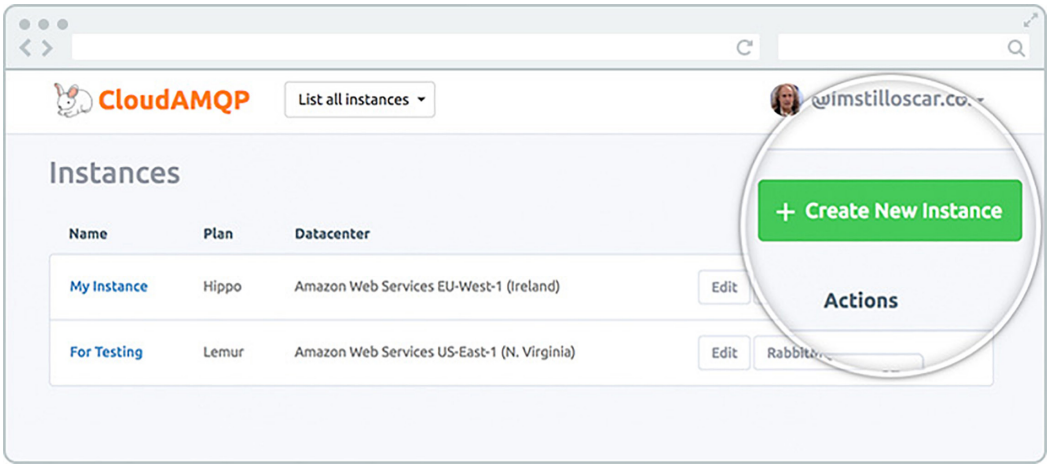


***Figure 6** - Instances in the CloudAMQP web interface.*

It is possible to send messages across languages, platforms, and operating systems once the RabbitMQ instance is up and running. Start by opening the management interface to get an overview of the RabbitMQ server.

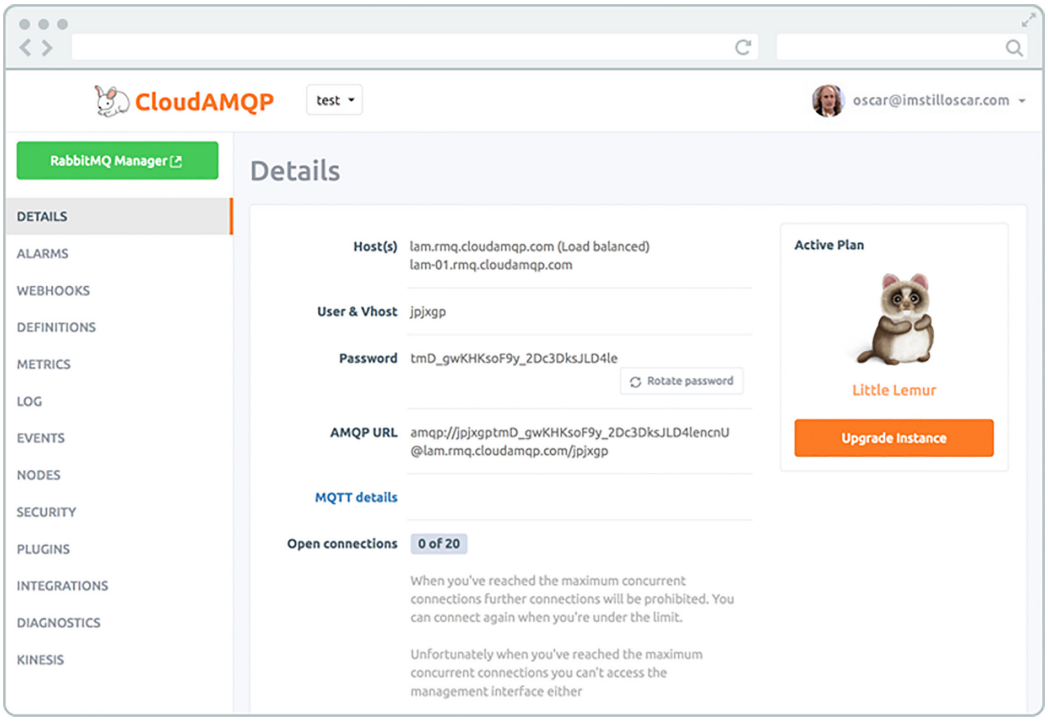


***Figure 7** - Detailed information of an instance in the CloudAMQP Console.*

The Management Interface

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

The management interface allows users to manage, create, delete, and list queues. It allows for monitoring of queue length, checking message rate, changing and adding user permissions, and much more. Detailed information about the management interface is provided in the chapter titled The Management Interface.

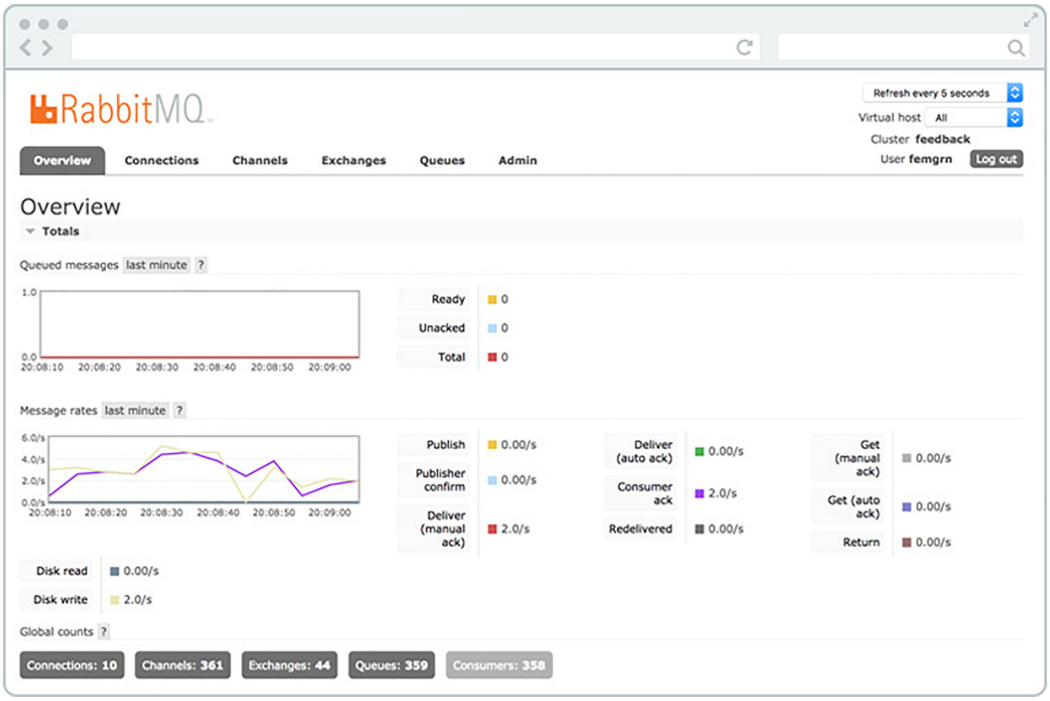


Figure 8 *- The overview window in the RabbitMQ management interface.*

Publish and Consume Messages

RabbitMQ speaks the AMQP protocol by default, and to be able to communicate with RabbitMQ, a library that understands the same protocol should be used. A RabbitMQ client library (sometimes called helper library) abstract the complexity of the AMQP protocol into simple methods; in this case, to communicate with RabbitMQ. The methods should be used when connecting to the RabbitMQ broker using the given parameters, hostname, port number, etc., for example, or when declaring a queue or an exchange. There is a choice of libraries for all major programming languages.

1. First of all, we need to set up/create a connection object. Here, the username, password, connection URL, port, etc., are 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. Remember, it is possible to use different programming languages in different parts of the system. The publisher could be written in Node.js while the subscriber is written in Python, for example.

Hint: Separate Projects and Environments Using Vhosts

Like it's possible to create different databases within a PostgreSQL (database) server for different projects, vhost makes it possible to separate applications on one single broker. Isolate users, exchanges, queues, etc. to one specific vhost or 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 is no resource isolation between vhosts. Shared plans on CloudAMQP are located on isolated vhosts.

PART ONE

EXCHANGES, ROUTING KEYS AND BINDINGS

What are 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 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 message routing agents, living in a virtual host (vhost) within RabbitMQ. Exchanges accept messages from the producer application and route them to message queues with the help of header attributes, bindings, and routing keys.

A binding is a "link" configured to make a connection between a queue and an exchange. The routing key is a message attribute. The exchange might look at the routing key, depending on exchange type, when deciding on how to route the message to the correct queue.

Exchanges, connections, and queues can be configured to include properties such as durable, temporary, and auto delete upon creation. Durable exchanges survive server restarts and last until they are 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 own unique exchanges or use the predefined default exchanges.

DIRECT EXCHANGE

A direct exchange delivers messages to queues based on a routing key. The routing key is a message attribute added to the message by the producer. Think of the routing key as an "address" that the exchange uses to decide on how to route the message. **A message goes to the queue(s) that have or has the exact match in binding key to the routing key of the message.** The direct exchange type is useful 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 the 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).

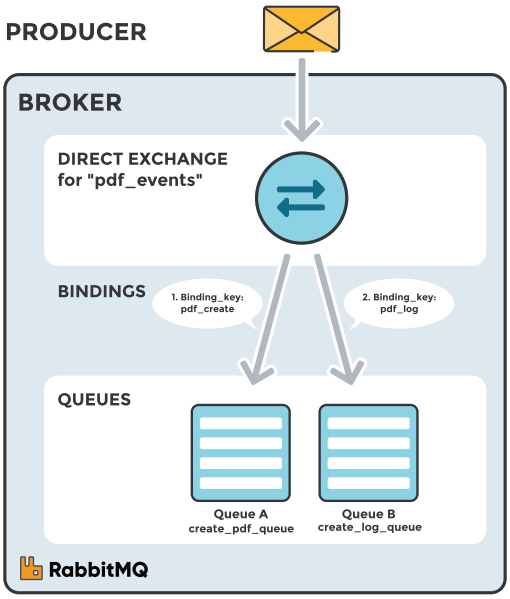


Figure 9 - A message is directed to the queue where the binding key is an exact match of the message's routing key.

Scenario 1

- Exchange: pdf_events
- Queue A: create_pdf_queue
- 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

A message with the routing key pdf_log is sent to the exchange pdf_events (Figure 9). The message is routed to create_log_queue because 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 using the default exchange, the 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 that matches the queue name.

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 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*).

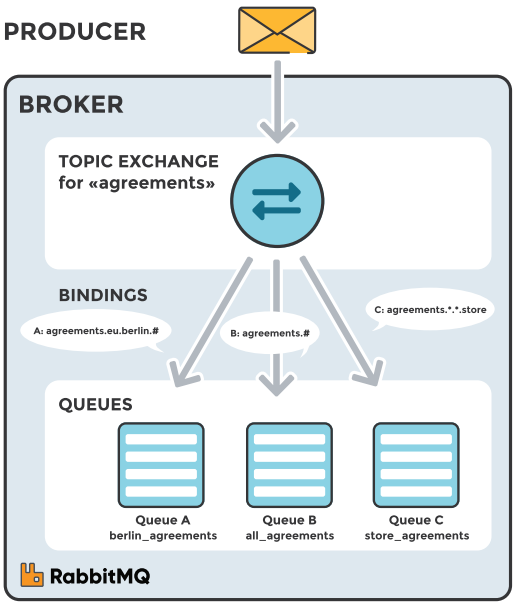


Figure 10 - Messages are routed to one or many queues based on a match between a message routing key and the routing patterns.

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 handles the message.

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

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 (berl in_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*

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

Fanout exchanges copy and route 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 distributed systems designed 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 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)

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.

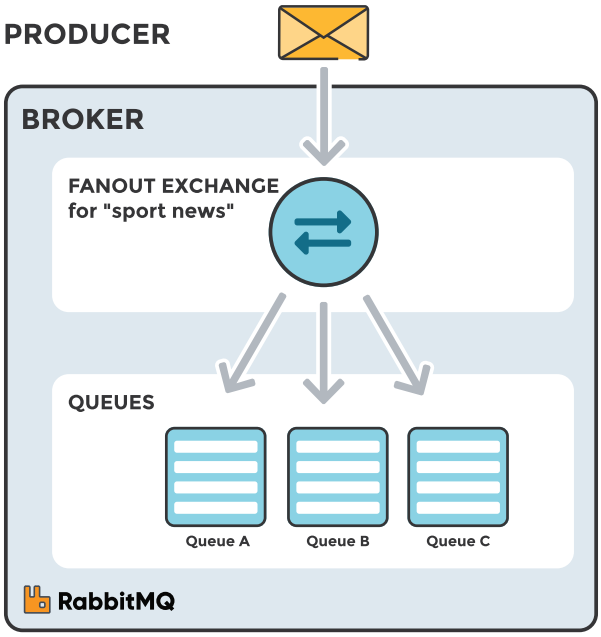


Figure 11 *- Fanout Exchange: The received message is routed to all queues that are bound to the exchange.*

HEADERS EXCHANGE

A Headers exchange routes messages based on arguments containing headers and optional values. Similar to topic exchanges, headers exchanges decide their routes based on header values instead of routing keys. A message is considered a match if the value of the header equals the value specified on the binding.

A unique argument named “x-match” can be added in the binding between exchange and the message queue. This argument decides if all headers must match or just one. Either common headers between the message and the binding counts as a match, or all the headers referenced in the binding need to be present in the m essage 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. Integer and hash functions are preferred over commonly used strings. 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”.

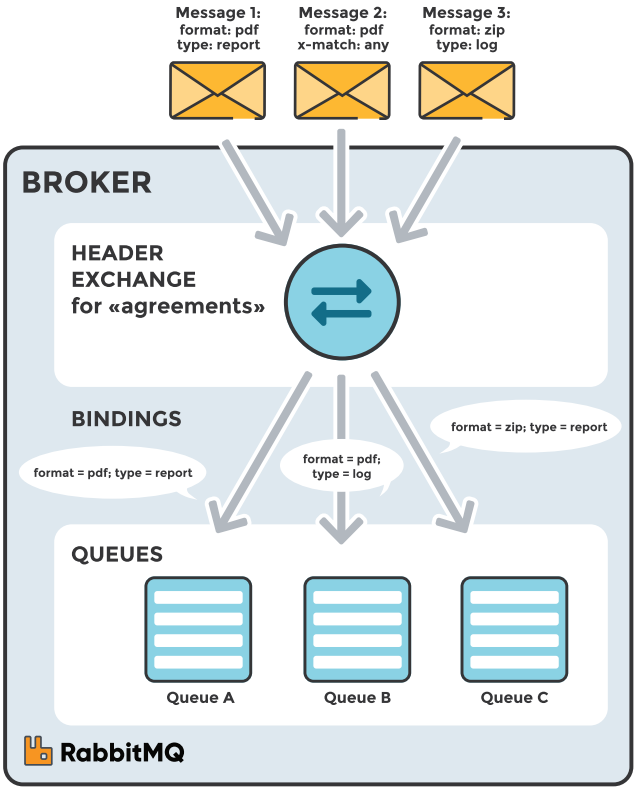


Figure 12 *– Headers exchange routes messages to queues that are bound using arguments (key and value) containing headers and optional values.*

- 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 type)

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 type)

DEAD LETTER EXCHANGE

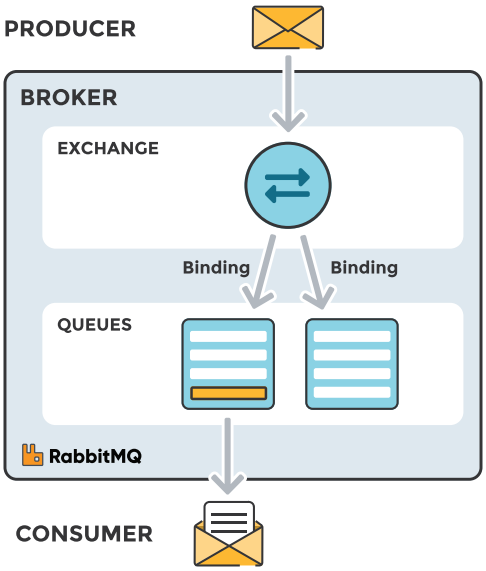
RabbitMQ provides an AMQP extension known as the dead letter exchange. The dead letter exchange captures messages that are not deliverable and routes them out of the queue. A message is considered "dead" when it has reached the end of it's time-to-live, the queue exceeds the max length (messages or bytes) configured for the queue, or the message has been rejected by the queue or nacked by the consumer for some reason and is not marked for re-queueing.

PART ONE

RABBITMQ AND CLIENT LIBRARIES

A client library that understands the same protocol is needed to communicate with RabbitMQ. Fortunately, there are many options for AMQP client libraries in many different languages, and those client libraries have several methods to communicate with a 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, generates a PDF, and emails it back to the user. A RabbitMQ instance is required in order to practice the coding examples.



PART ONE

RABBITMQ WITH RUBY AND BUNNY

Start by downloading the RabbitMQ client library for Ruby. Ruby developers have a number of options for AMQP client libraries. In this example is Bunny (http://rubybunny.info/) used, an asynchronous client for publishing and consuming of messages.

When running the full *example_publisher.rb* code below, a connection is established between the RabbitMQ instance and the 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.

FULL TUTORIAL SOURCE CODE

```
# example_publisher.rb

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)

q.bind(x, :routing_key => "process")

# Publish a message

information_message = "{\"email\":\"example@mail.com\",\"name\": \"name\",\"size\":
\"size\"}"
```

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

sleep 1.0

conn.close


# example_consumer.rb

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 => true) do |delivery_info, properties, payload|

  json_information_message = JSON.parse(payload)

  pdf_processing(json_information_message)

end
```

PUBLISHER

```
# example_publisher.rb

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) is created in the TCP connection.Publishing and consuming of messages, and subscribing to a queue, is all done over a channel.

Declare a queue

```
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

```
# 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.

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: https://www.cloudamqp.com/docs/ruby.html

PART ONE

RABBITMQ AND NODE.JS WITH AMQPLIB

Start by downloading the client library for Node.js. Node developers have a number of options for AMQP client libraries. In this example, amqplib (https://www.npmjs.com/package/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 rouhng_key , if it exists. (The default exchange is a direct exchange with no name)

The full code can be downloaded from CloudAMQP documentation pages.

FULL TUTORIAL SOURCE CODE

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


// if the connection is closed or fails to be established, it will reconnect
var amqpConn = null;

function start() {

  amqp.connect(process.env.CLOUDAMQP_URL + "?heartbeat=60", function(err, conn) {

    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();
  });
}
function whenConnected() {
  startPublisher();
  startWorker();
}
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 m = offlinePubQueue.shift();
      if (!m) break;
      publish(m[0], m[1], m[2]);
    }
  });
}
// method to publish a message, will queue messages internally if the connection is
down and resend later
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]);
  }
}
// A worker that acks messages only if processed succesfully
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);
        }
      });
    }
  });
}
function work(msg, cb) {
  console.log("Got msg ", msg.content.toString());
  cb(true);
}
function closeOnErr(err) {
  if (!err) return false;
  console.error("[AMQP] error", err);
  amqpConn.close();
  return true;
}

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

start();
```

PUBLISHER

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

Set up a connection

```
// if the connection is closed or fails to be established, it will reconnect
var amqpConn = null;
function start() {
```

```
  amqp.connect(process.env.CLOUDAMQP_URL + "?heartbeat=60", function(err, conn) {
    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 establishes 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 m = offlinePubQueue.shift();
      if (!m) break;
      publish(m[0], m[1], m[2]);
    }
  });
}
```

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

```
// method to publish a message, will queue messages internally if the connection is
down and resend later
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 publishes 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 succesfully
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("Got msg ", 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;
}
```

Trigger Publish function

```
setInterval(function() {
  publish("", "jobs", new Buffer.from("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.

PART ONE

RABBITMQ AND PYTHON WITH PIKA

Start by downloading the client-library for Python3. The recommended library for Python is Pika (https://pika.readthedocs.io/en/stable/). Put *pika==1.1.0* 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 TUTORIAL SOURCE 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

connection = pika.BlockingConnection(params)
channel = connection.channel()
channel.queue_declare(queue='pdfprocess')
# send a message
channel.basic_publish(exchange='', routing_key='pdfprocess', body='User information')
print ("[x] Message sent to consumer")
connection.close()

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

```
def pdf_process_function(msg):
  print(" PDF processing")
  print(" [x] Received " + str(msg))

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

# Access the CLODUAMQP_URL environment variable and parse it (fallback to localhost)
url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost:5672/%2f')
params = pika.URLParameters(url)
connection = pika.BlockingConnection(params)
channel = connection.channel() # start a channel
channel.queue_declare(queue='pdfprocess') # Declare a queue

# 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('pdfprocess',
  callback,
  auto_ack=True)

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

PUBLISHER

```
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
```

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

```
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 information')

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

```
def pdf_process_function(msg):
  print(" PDF processing")
  print(" [x] Received " + str(msg))
  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.

Consume

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

basic_consume binds messages to the consumer callback function.

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

PART ONE

THE MANAGEMENT INTERFACE

Queues, connections, channels, exchanges, users, and user permissions are handled - created, deleted and listed - from a web browser through the RabbitMQ management interface plugin. Additionally, message rates can be monitored and sending or receiving messages can be manually performed via the interface. The interface plugin consists of a single static HTML page that makes background queries to the HTTP API for RabbitMQ. The management interface is useful for debugging applications or an overview of the whole system is required.

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

All tabs in the management menu are explained in this chapter, which includes screenshots of overview, connections, and channels along with exchanges, queues, and admin including users and permissions. A simple example will also demonstrate how to set up a queue with an exchange and add a binding between them.

Concepts

- **Cluster** - A cluster consists of a set of nodes, i.e. connected computers, working together. A RabbitMQ instance consisting of more than one node is called a RabbitMQ cluster.
- **Node** - A node is a single computer in the RabbitMQ cluster.

OVERVIEW

The overview in Figure 13 shows two graphs; one graph for queued messages and one with the message rate. The time interval shown in the graph can be changed by pressing the text last minute above the graph. Information about all different statuses for messages can be found by pressing “?”.

Queued messages

This graph shows the total number of queued messages for all 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.

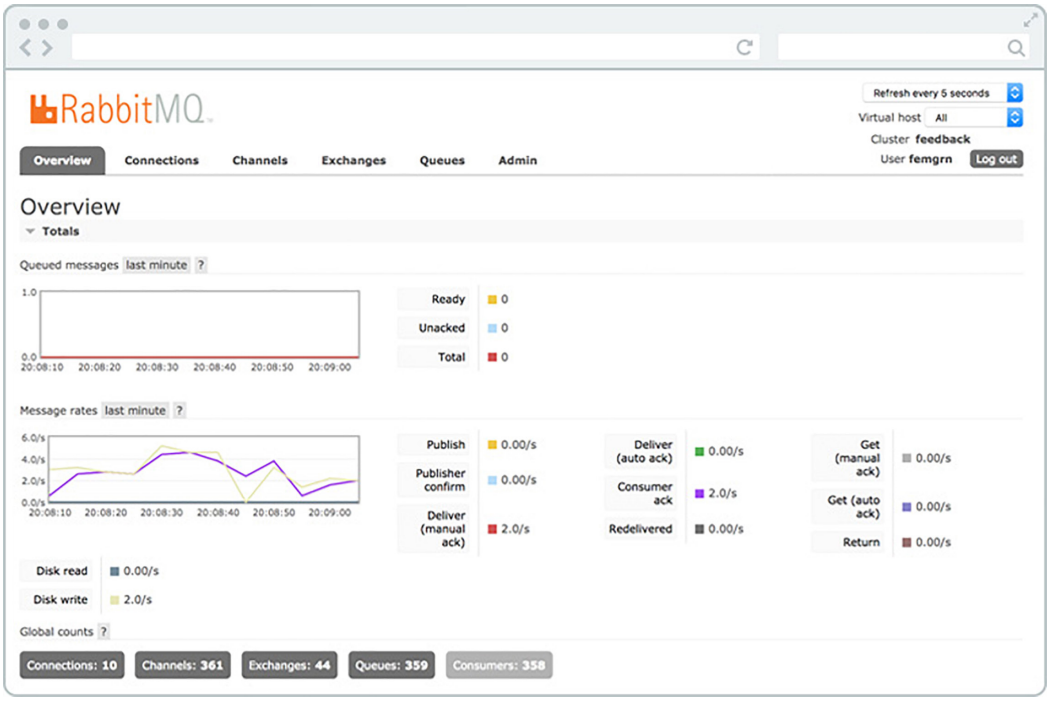


Figure 13 *- The RabbitMQ management interface.*

Message rate

Message rate is a graph with the rate of how fast the messages are being 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 to which the current user has access.

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. There is also 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.

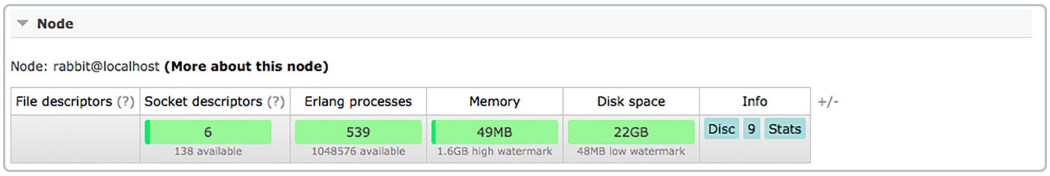


Figure 14 *– Node-specific information.*

Import/export definitions

It is possible to import and export configuration definitions. When downloading the definitions, a JSON representation of the broker (the RabbitMQ settings) is given. This JSON can be used to restore exchanges, queues, vhosts, policies, and users. This feature can be used as a backup solution

CONNECTIONS AND CHANNELS

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

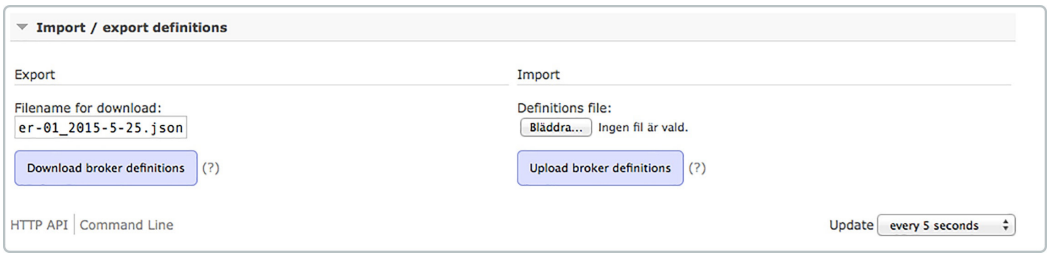


Figure 15 *– Import/export definitions as a JSON file.*

Connections

The connections tab (Figure 16) 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 displays the number of channels using the connection. SSL/TLS indicate whether the connection is secured with SSL or not.

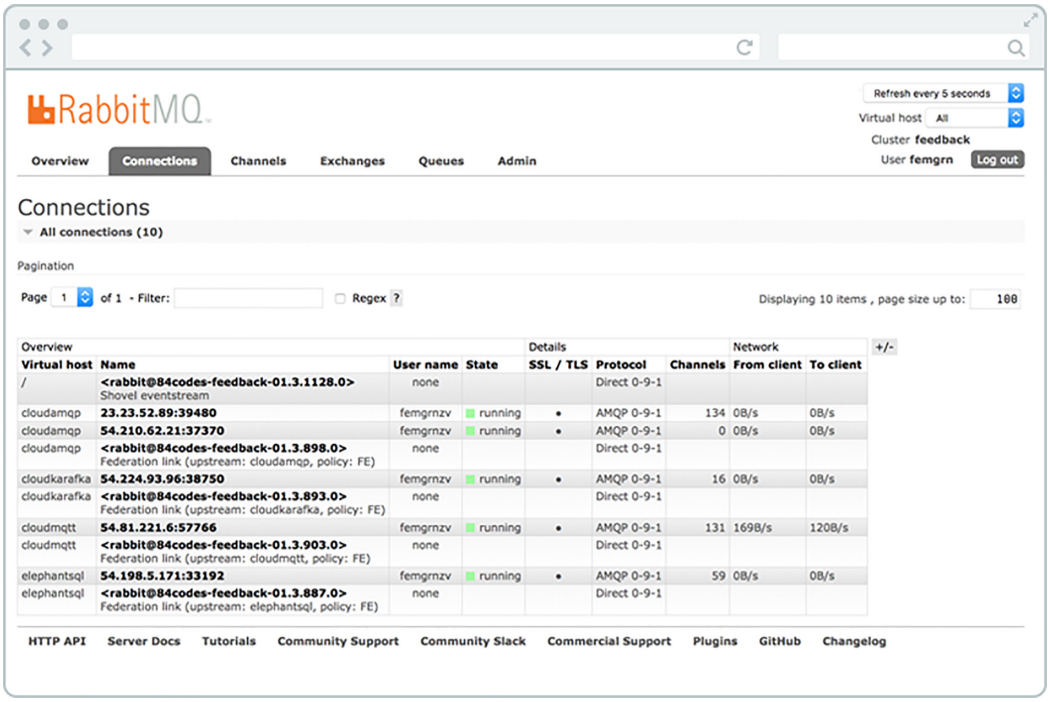


Figure 16 *– The Connections tab in the RabbitMQ management interface.*

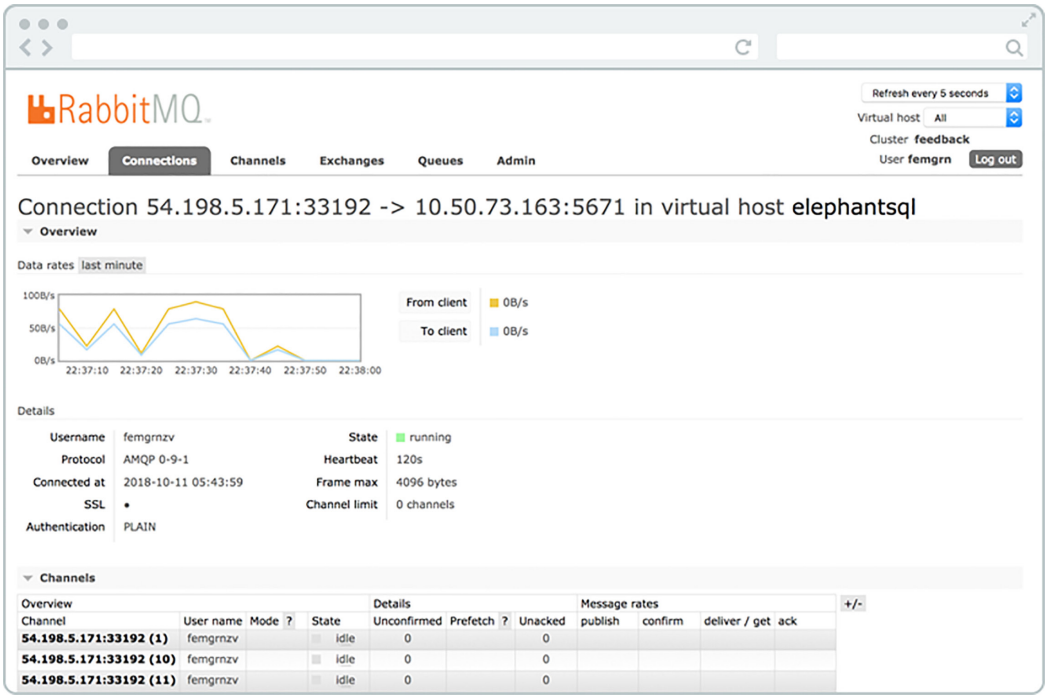


Figure 17 *- Connection information for a specific connection.*

Clicking on one of the connections gives an overview of that specific connection (Figure 17) including the channels within the connection and the data rates as well as client properties, and the connection can be closed here as well.

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.

Channels

The channel tab (Figure 18) 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.

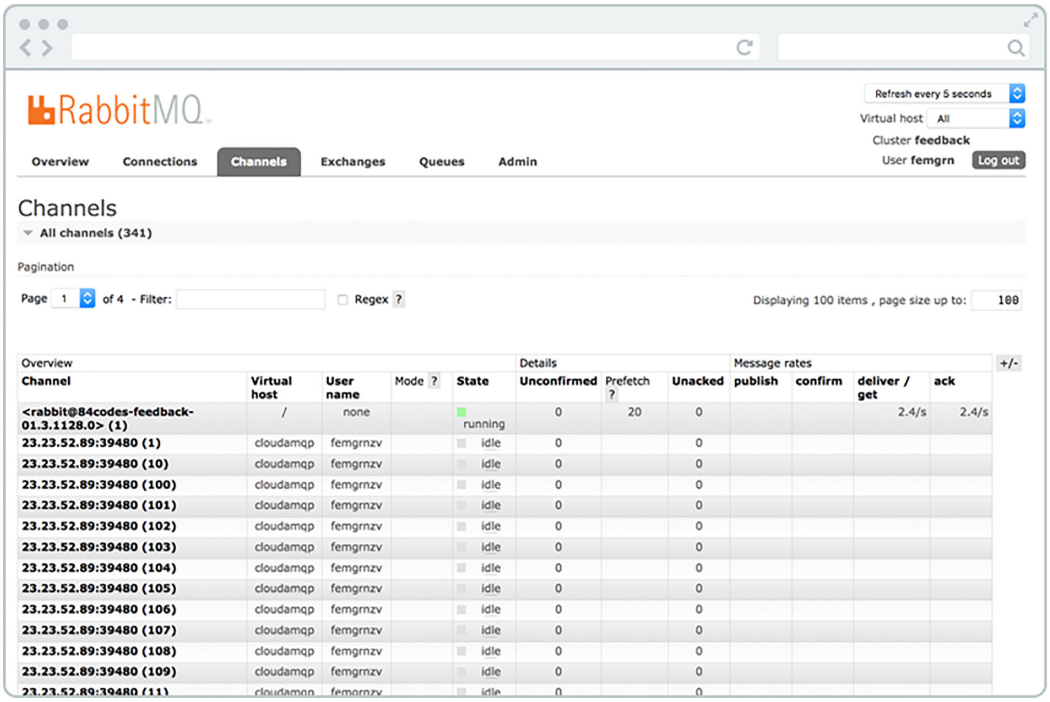


Figure 18 – The Channels tab

Clicking on one of the channels provides a detailed overview of that specific channel (Figure 19). The message rate and the number of logical consumers retrieving messages from the channel are also displayed.

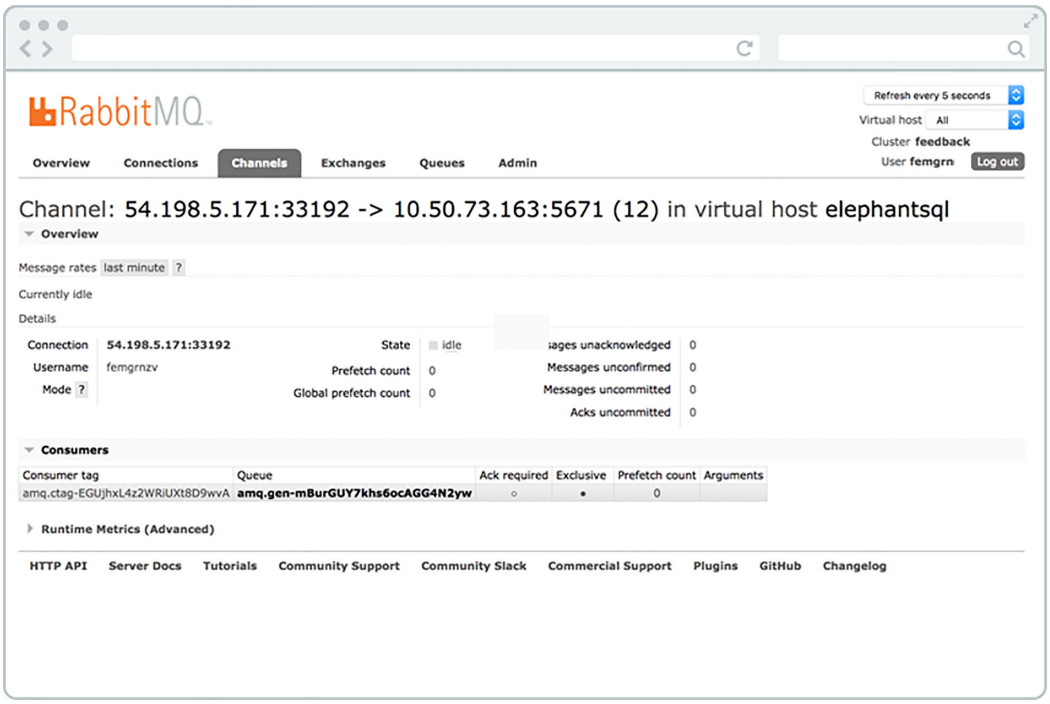


Figure 19 *- Detailed information about a specific channel.*

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

EXCHANGES

All exchanges can be listed from the exchange tab (Figure 20). 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, which are created by default.

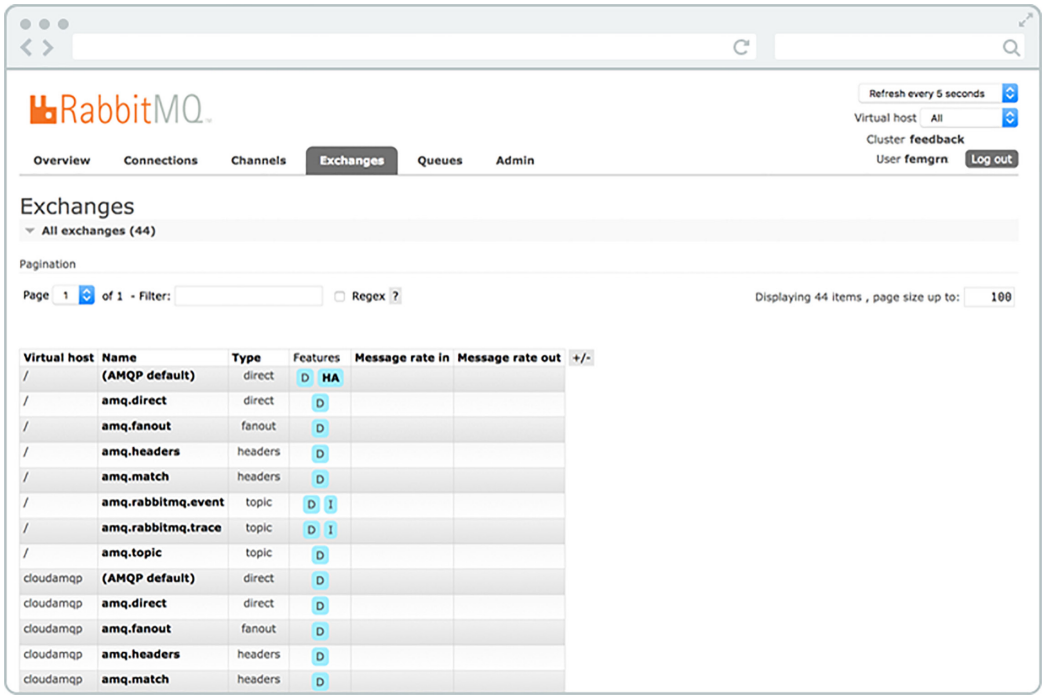


Figure 20 *- The exchanges tab in the RabbitMQ management interface.*

Figure 21 *- Detailed view of an exchange.*

Clicking on the exchange name displays a detailed page about the exchange (Figure 21). Adding bindings to the exchange and viewing already existing bindings is are also actions that can be taken in this area as well as publishing a message to the exchange or deleting the exchange.

QUEUES

The queue tab shows the queues for all or one selected vhost (Figure 22). Queues may also be created from this area. Queues have different parameters and arguments depending on how they were created. The features column shows the parameters that belong to the queue, such as:

- **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 being discarded.
- **Auto-expire** - The time a queue can be unused before it is automatically deleted.
- **Max length** - How many ready messages a queue can hold before it starts to drop them.
- **Max length bytes** - The total size of ready messages a queue can hold before it starts to drop them.

Clicking on any chosen queue from the list of queues will show all information about it, as shown in Figure 23.

The first two graphs include the same information as the overview, however, they show the number and rates for this specific queue.

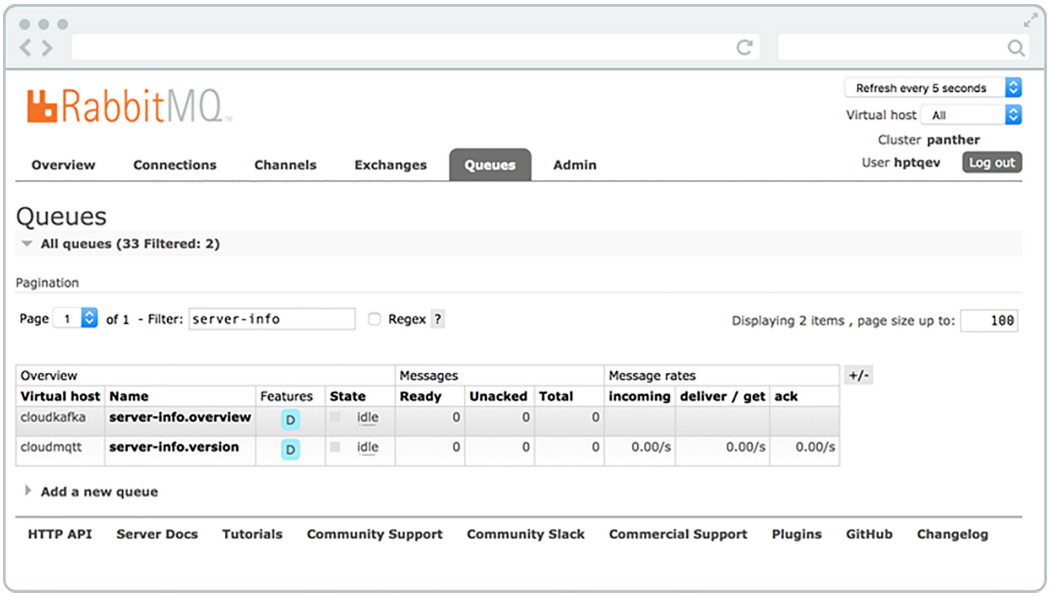


***Figure** 22 - The queues tab in the RabbitMQ management interface.*

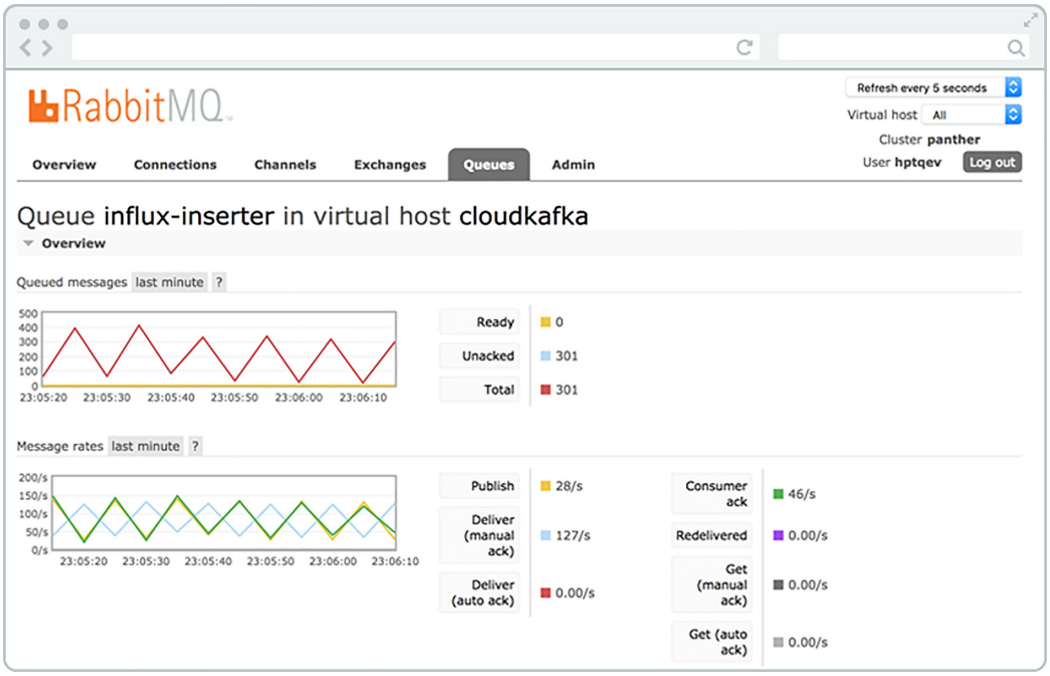


***Figure 23** - Specific information about a single queue.*

CONSUMERS

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

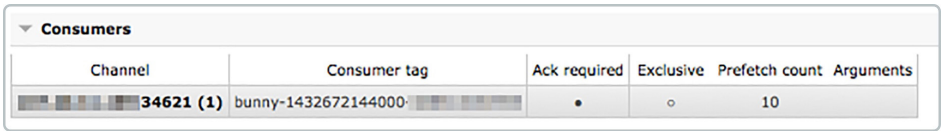


***Figure 24** - Consumers connected to a queue.*

Bindings

All active bindings to the queue are shown under bindings. New bindings to queues can be created from here or unbinding a queue from an exchange (Figure 25) as well.

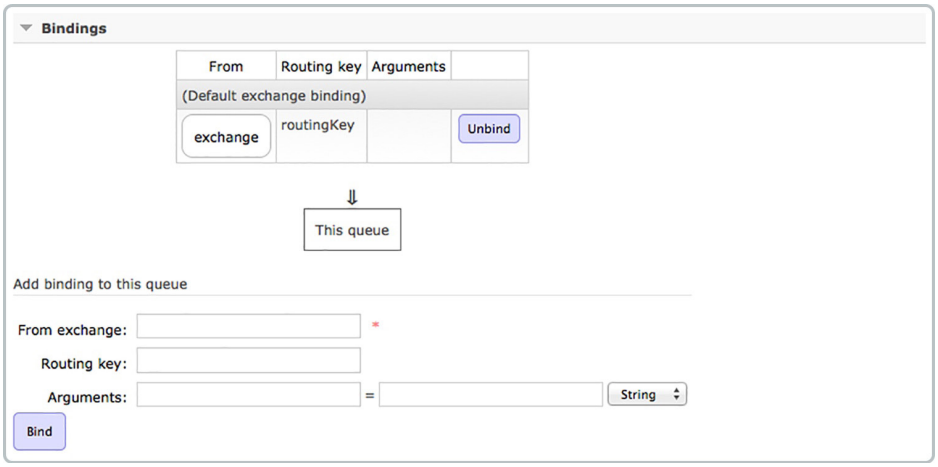


Figure 25 *- The bindings interface.*

Publish message

Publishing a message can be performed manually to the queue from this area. 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 proper queue. It is also possible to publish a message to an exchange from the exchange view.

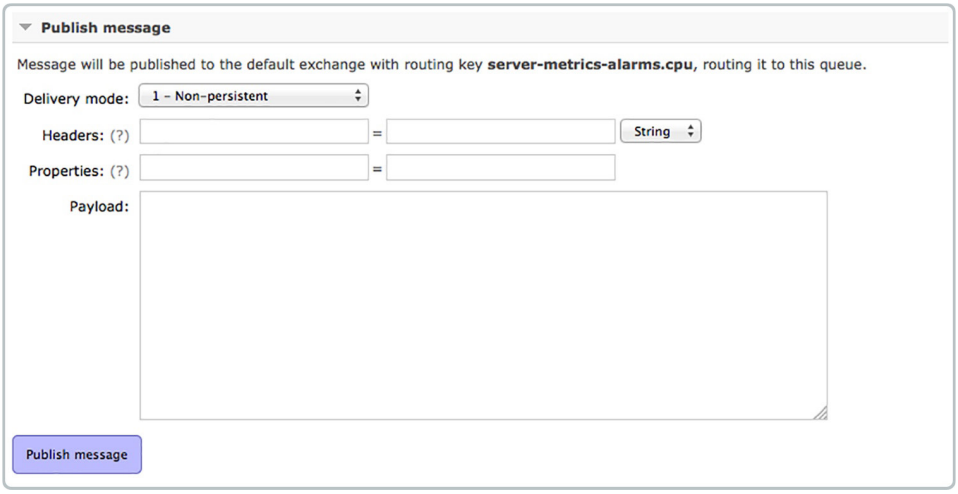


Figure 26 *- Manually publishing a message to the queue.*

Get message

Manually inspecting the message in the queue can be done in this area. Get message gets the message available for review, while marking it as re-queue will cause RabbitMQ to place it back in the queue in the same order.

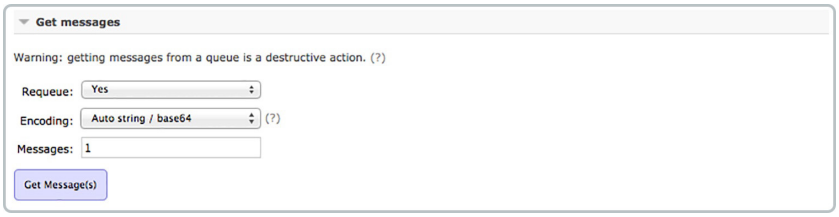


***Figure 27** - Manually inspect a message.*

Delete or Purge queue

A queue can be deleted by pressing the delete button or the queue can be emptied with use of the purge function.

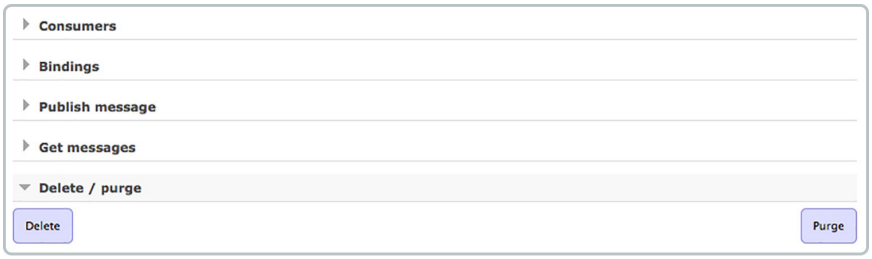


***Figure 28** - Delete or purge a queue from the web interface.*

ADMIN

The Admin view (Figure 29) is where users are added and permissions for them are changed. This area is also used to set up vhosts (Figure 30), policies, federation, and shovels. Information about shovels can be found here: https://www.rabbitmq.com/shovel.html while information about federation will be given in part two of this book.

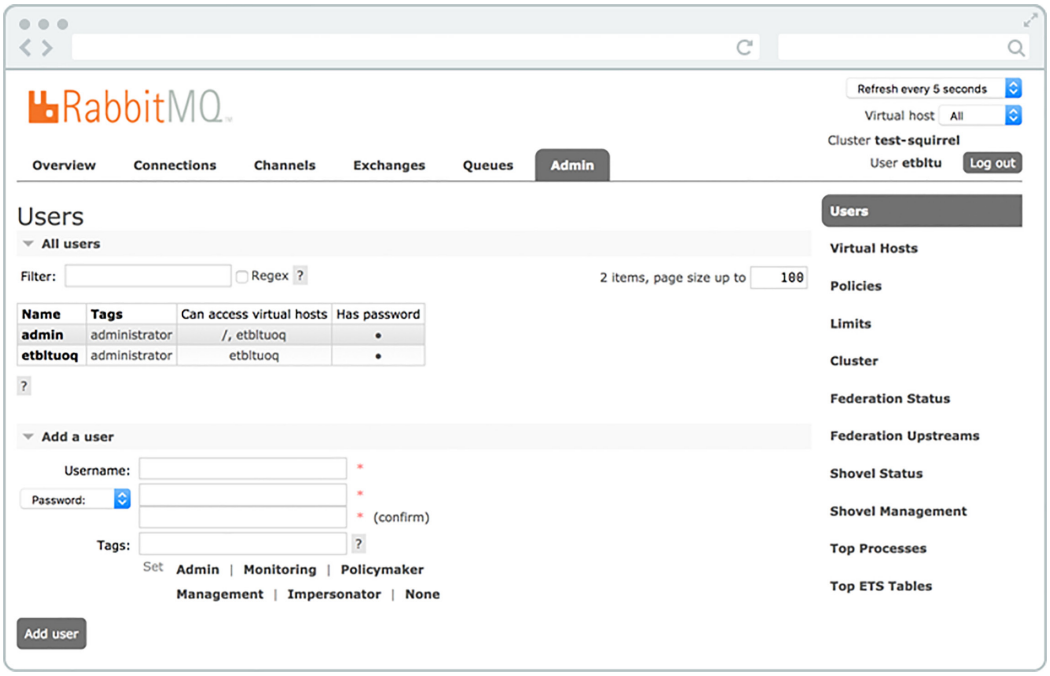


***Figure 29** - The Admin interface where users can be added.*

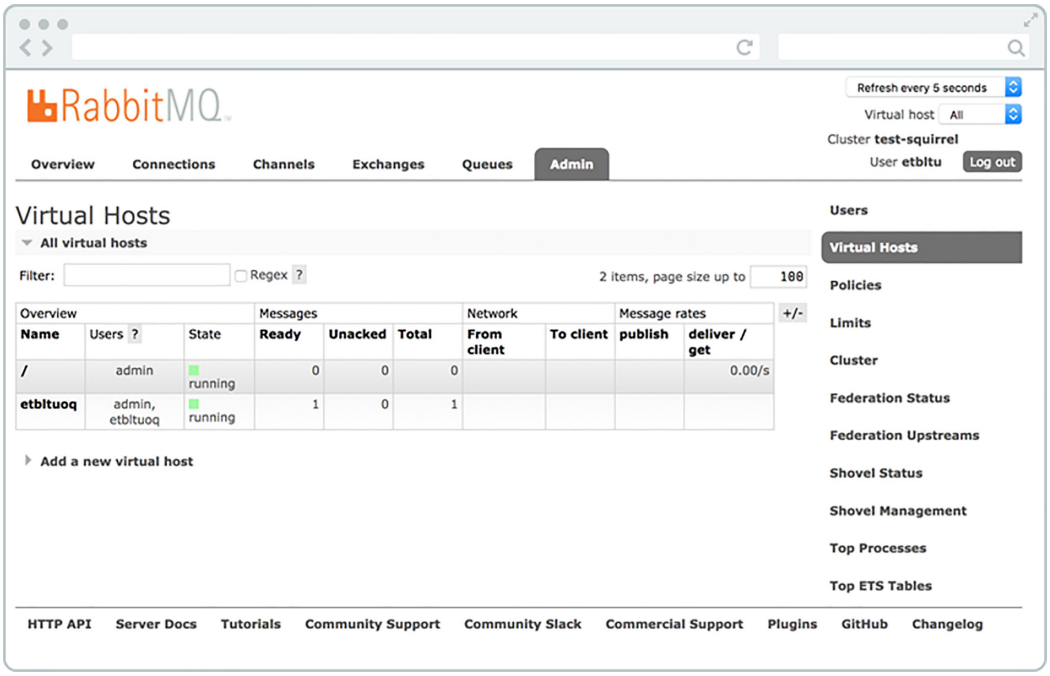


***Figure 30** - Virtual Hosts can be added from the Admin tab.*

The example in Figure 31, shows how to create an example-queue and an exchange called example.exchange (Figure 32).

The exchange and the queue are connected by a binding called pdfprocess (Figure 33). Messages can be published (Figure 34) to the exchange with the routing key pdfprocess, and will end up in the queue (Figure 35).

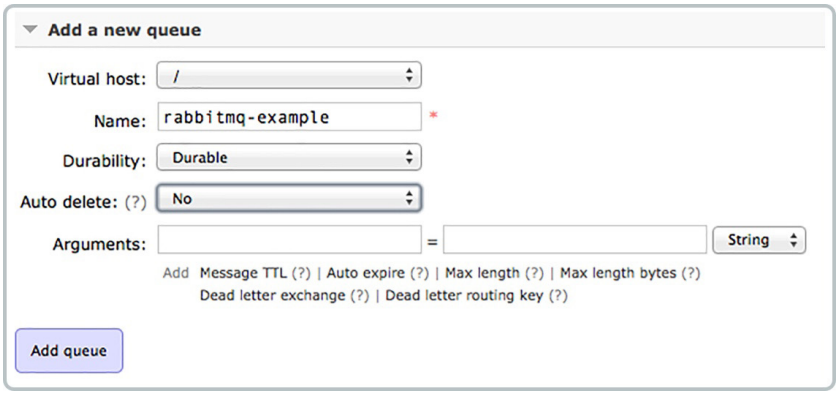


***Figure 31** - Queue view, add queue.*

Many things are viewed and handled via the management interface, and studying it will give an overview of the system and the relationship between functions.

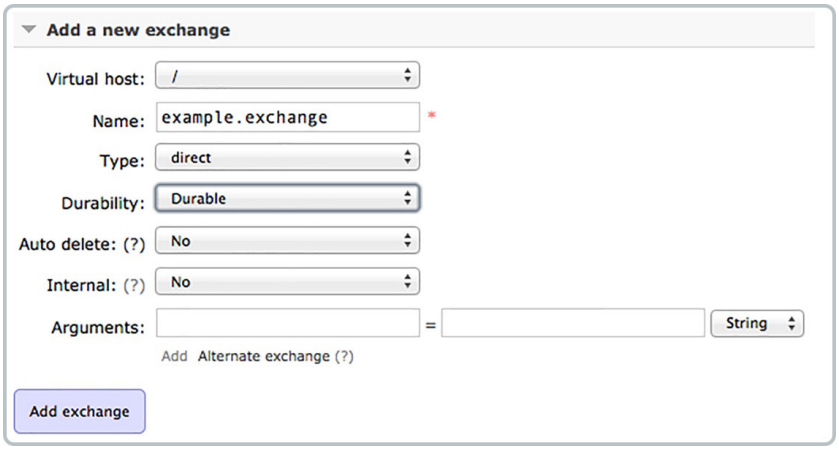


***Figure 32** - Exchange view, add exchange.*

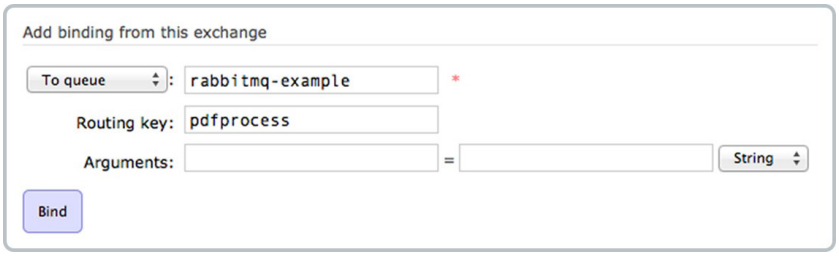


***Figure 33** - Click on the exchange or on the queue, go to "Add binding from this exchange" or "Add binding to this queue".*

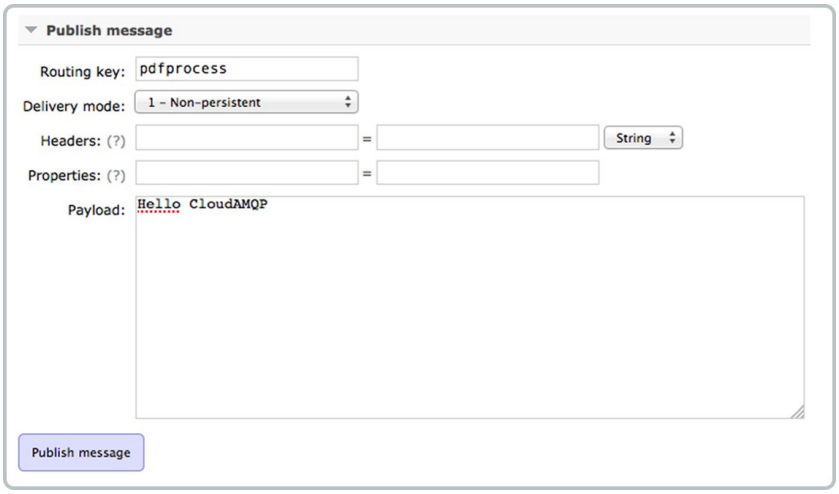


Figure 34 *- Publish a message to the exchange with the routing key "pdfprocess".*

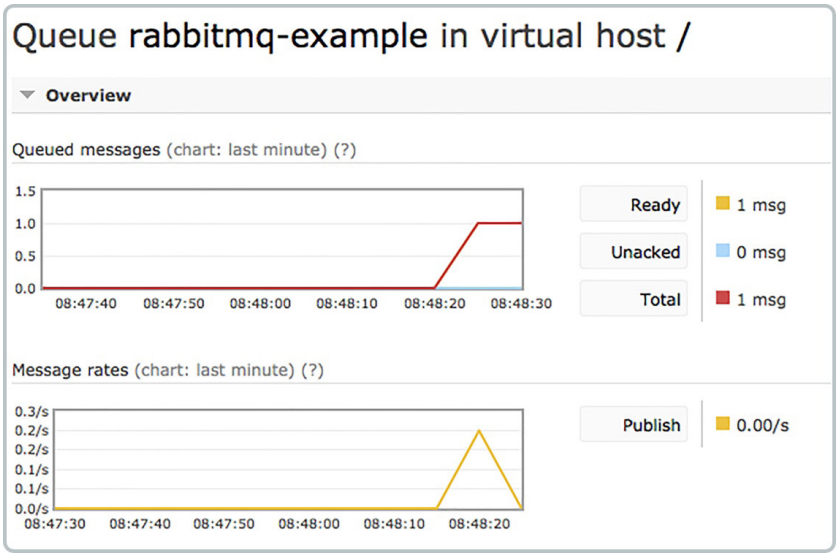


Figure 35 *- Queue overview for example-queue when a message is published.*

PART TWO

ADVANCED MESSAGE QUEUING

BEST PRACTICE

Various configurations affect your RabbitMQ cluster in different ways. Learn how to optimize performance through the setup.

Some applications require high throughput while other applications are publishing batch jobs that can be delayed for a while. Tradeoffs must be accepted between performance and guaranteed message delivery. The goal when designing the system should be to maximize combinations of performance and availability that make sense for the specific application. Bad architectural design decisions and client-side bugs can damage the broker or affect 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 CloudAMQP gives some extra attention such as how to migrate a cluster with queue federation.

PART TWO

RABBITMQ BEST PRACTICE

These are the RabbitMQ Best Practice recommendations based on the experience CloudAMQP have gained while working with RabbitMQ. This includes information on how to configure a RabbitMQ cluster for optimal performance and how to configure it to get the most stable cluster. Queue size, common mistakes, lazy queues, pre-fetch values, connections and channels, HiPE, and number of nodes in a cluster are some of the things discussed.

QUEUES

Performance optimizations of RabbitMQ queues.

Keep queues short, if possible

If suitable for use, keep 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 it (a persistent message in a durable queue will go to disk). The recommendation is to keep fewer 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.

Keep queues short

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 is time-consuming to restart a cluster with many messages because the index must be rebuilt. It takes time to sync messages between nodes in the cluster after a restart.

Enable lazy queues to get predictable performance

Queues can become long for various reasons; consumers might crash, or be offline due to maintenance, or they might simply be working slower than usual. Lazy queues are able to support long queues (millions of messages). Lazy queues, added in RabbitMQ 3.6, write messages to disk immediately, thus spreading the work out over time instead of taking the risk of a performance hit somewhere down the line. It provides a more predictable, smooth performance without any sudden drops, but at the cost. Messages are only loaded into memory when needed, thereby minimizing the RAM usage, but increasing the throughput time.

Sending many messages at once (e.g., processing batch jobs) or if there is a risk consumers will not keep up with the speed of the publishers consistently, lazy queues are recommended. Disable lazy queues if high performance is required, if queues are always short, or if a max-length policy exists.

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

Limit queue size with TTL or max-length

Another recommendation 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 never becomes larger than the max-length setting.

Number of queues

Queues are single-threaded in RabbitMQ, with one queue able to handle up to about 50,000 messages. Better throughput is achieved on a multi-core system if multiple queues and multiple consumers are used. Optimal throughput is achieved with 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 there are thousands upon thousands of active queues and consumers. The CPU and RAM usage may also be affected negatively with the use of too many queues.

Split queues over different cores

Queue performance is limited to one CPU core or hardware thread because a queue is single threaded. Better performance is the result if queues are split over different cores and into different nodes when using a RabbitMQ cluster. Routing messages to multiple queues results in a much higher overall performance.

RabbitMQ queues are bound to the node where they are first declared. All messages routed to a specific queue will end up on the node where that queue resides. Manually splitting queues evenly between nodes is possible, but the downside is to remember where each queue is located manually.

Two plugins that help if there are multiple nodes or a single node cluster with multiple cores are the consistent hash exchange plugin and RabbitMQ Sharding.

Use multiple queues and consumers

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

The consistent hash exchange plugin

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

The consistent hash exchange plugin is used to get maximum use of many cores in the cluster. Note that it is 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 partitions queues automatically after an exchange is defined as sharded. The supporting queues are then automatically created on every cluster node and messages are sharded across them. RabbitMQ sharding shows one queue to the consumer, but many queues could be running behind it in the background. The RabbitMQ sharding plugin is a centralized place to send messages with a goal of load balancing through the addition of queues to the other nodes in the cluster. Read more about RabbitMQ sharding here: https://github.com/rabbitmq/rabbitmq-sharding.

Don't set names on temporary queues

Setting a queue a name is important for sharing the queue between producers and consumers, but it is not when using temporary queues. Instead, allow the server to choose a random queue name, 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 queues behind. Leaving too many queues behind might affect the performance of the system. There are three ways to have queues deleted automatically.

The first option is to set a **TTL (time-to-live) policy** on the queue. A TTL policy of 28 days will delete queues that have not had messages consumed from them in the last 28 days.

Another option is an **auto-delete** queue, which 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).

Finally, an **exclusive queue** is one that is only used (consumed from, purged, deleted, etc.) by its declaring connection. Exclusive queues are deleted when their declaring connection is closed or gone (due to underlying TCP connection loss or other circumstances).

Set limited use on priority queues

Queues can have zero or more priority levels. Keep in mind that each priority level uses an internal queue on the Erlang VM, which means that it takes up resources. In most use cases, it is sufficient to have no more than five priority levels, which keeps resource use manageable.

Payload - RabbitMQ Message Size and Type

How to handle the payload size of messages sent to RabbitMQ is a common question by developers. Avoid sending very large files in messages, but also keep in mind that the rate of messages/second can be a larger bottleneck than the message size itself. Sending multiple small messages might be a bad alternative. The better idea could be to bundle the small messages into one larger message and let the consumer split it up. However, if bundling multiple messages remember that this might affect the processing time. If one of the bundled messages fails, will all of them need to be re-processes? Bandwidth and architecture will dictate the best way to set up messages to considerd payload.

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 a worst case scenario, the server can crash due to running out of memory. Try to keep connection/channel count low.

Don't use too many connections or channels

Try to keep connection/channel count low.

Don't share channels between threads

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).

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

Don't open and close connections or channels repeatedly, as doing so will create a higher latency because more TCP packages have to be sent and received.

The handshake process for an AMQP connection is actually quite involved and requires at least seven TCP packets (more if TLS is used). The AMQP protocol has a mechanism called channels that "multiplexes" a single TCP connection. It is recommended that each process only create one TCP connection with multiple channels in that connection for different threads. Connections should be long-lived so that channels can be opened and closed more frequently, if required. Even channels should be long-lived if possible. Do not open a channel every time a message is published. Best practice includes the re-use of connections and multiplexing a connection between threads with channels, when possible.

- 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)

Don't open and close connections or channels repeatedly

Don't open and close connections or channels repeatedly - doing that means a higher latency, as more TCP packages have to be sent and received.

Separate connections for publisher and consumer

Unless the connections are separated between publisher and consumer, messages may not be consumed. This is especially true if the connection is in flow control, which will constrict the message flow even more.

Another thing to keep in mind is that RabbitMQ may cause back pressure on the TCP connection when the publisher is sending too many messages to the server. When consuming on the same TCP connection, the server might not receive the message acknowledgments from the client, affecting the performance of message consumption and the overall server speed.

Separate connections for publisher and consumer

For the highest throughput, separate the connections for publisher and consumer.

RABBITMQ MANAGEMENT INTERFACE PLUGIN

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, which consumes server resources.

ACKNOWLEDGMENTS AND CONFIRMS

Messages in transit might get lost in an event of a connection failure and may need to be retransmitted. Acknowledgments let the server and clients know when to retransmit messages. The client can either ack(knowledge) the message when it receives it, or when the client has completely processed the message. Acknowledgment has a perfor-

mance 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.) do not go missing.

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

UNACKNOWLEDGED MESSAGES

All unacknowledged messages must reside in RAM on the servers. Too many unacknowledged messages will eventually use all system memory. An efficient way to limit unacknowledged messages is to limit how many messages the clients pre-fetch. Read more about this in the pre-fetch section.

PERSISTENT MESSAGES AND DURABLE QUEUES

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

Queues should be declared as "durable" and messages should be sent with delivery mode "persistent".

Persistent messages are heavier as they have to be written to disk. Similarly, lazy queues have the same effect on performance, even though they are transient messages. For high performance, use transient messages; for high throughput, use temporary or non-durable queues.

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

Connecting to RabbitMQ over AMQPS is the AMQP protocol wrapped in TLS. TLS has a performance impact since all traffic must be encrypted and decrypted. For maximum performance, use VPC peering instead, which encrypts the traffic without involving the AMQP client/server.

CloudAMQP configures the RabbitMQ servers so that they accept and prioritize fast but secure encryption ciphers.

PRE-FETCH

The pre-fetch value is used to specify how many messages are consumed at the same time. It is used to get as much out of the 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."

The RabbitMQ default pre-fetch setting gives clients an unlimited buffer, meaning that RabbitMQ by default sends 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 processed. Pre-fetch 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.

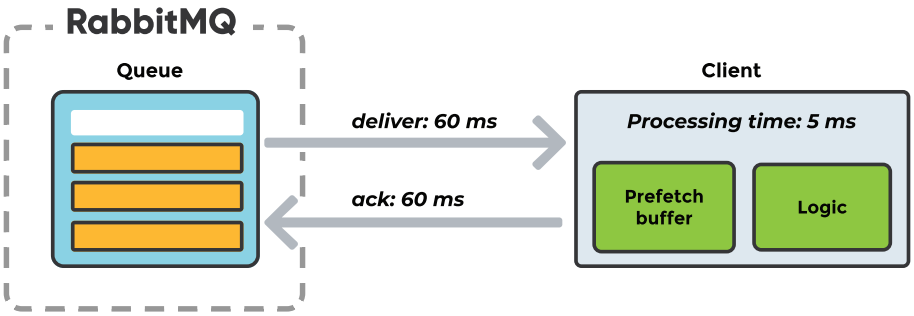


Figure 36 - Too small prefetch count may hurt performance

A pre-fetch count that is too small may hurt performance, as RabbitMQ typically waiting to get permission to send more messages. Figure 37 illustrates long idling time. In the example, we have a QoS pre-fetch setting of one (1). This means that RabbitMQ will not 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.

A large pre-fetch count, on the other hand, could deliver many messages to one single consumer, and keep other consumers in an idling state, as illustrated in Figure 37.

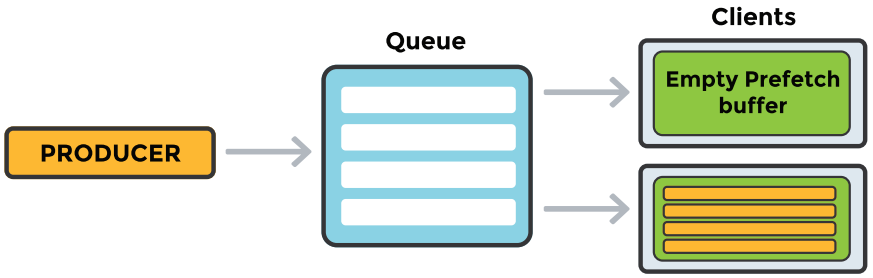


Figure 37 - A large prefetch count could deliver lots of messages to one single consumer.

How to set correct pre-fetch value

Messages from a single or a few consumers can be processed quickly, and pre-fetching many messages at once is a good practice as it keeps clients as busy as possible. If set to zero (0) and messages have about the same processing time all the time, with network behavior remaining the same, use the total round trip time/processing time on the client for each message to get an estimated pre-fetch value.

Messages from consumers and short processing times should be configured with a lower pre-fetch value. A too low value will keep the consumers idling too much as they wait for messages to arrive. A too high value may keep one consumer busy while other consumers are held in an idling state.

If the system has many consumers, and/or a long processing time, the best practice is to set the pre-fetch count to one (1) to evenly distribute the messages among all resources.

Please note that if a client is set up to auto-ack messages, the pre-fetch value has no effect.

The pre-fetch value will have no effect if the client auto-ack messages.

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

Don't use an unlimited pre-fetch 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. Enabling HiPE means that RabbitMQ is compiled at startup. The throughput increases from 20 to 80 percent according to benchmark tests. The drawback of HiPE is the startup time increases by about 1-3 minutes. HiPE is still marked as experimental in RabbitMQ's documentation.

CLUSTERING AND HIGH AVAILABILITY

Creating a CloudAMQP instance with one node results in one single high performance node, as the messages are not mirrored between multiple nodes.

Creating a CloudAMQP instance with two nodes results in 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 result in 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.

With three nodes created in a CloudAMQP instance, the result is one quarter 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. In the case of three nodes, pause minority is also a result, which means that shutting down the minority component reduces 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 on CloudAMQP clusters is that users create a new vhost but forget to enable an HA-policy for it. Messages will therefore not be synced between nodes.

ROUTING (EXCHANGES SETUP)

Direct exchanges are the fastest to use because multiple bindings mean that RabbitMQ must calculate where to send the message.

PLUGINS

Some plugins might consume a lot of CPU or use a large amount of RAM, which makes them less than ideal on a production server. Disable plugins that are not being used via the control panel in CloudAMQP.

Disable plugins that you are not using.

STATISTICS RATE MODE

The RabbitMQ management interface collects and stores stats for all queues, connections, channels, and more, which might affect the broker in a negative way if, for example, there are too many queues. Avoid setting the RabbitMQ management statistics rate to 'detailed' as it could affect performance.

Don't set management statistics rate mode as detailed.

RABBITMQ, ERLANG AND CLIENT LIBRARIES

Stay up-to-date with the latest stable versions of RabbitMQ and Erlang. CloudAMQP extensively tests new major versions before release; therefore, the most recommended version is the default in the dropdown menu for a new cluster.

Use the latest recommended version of client libraries and check the documentation or inquire directly with any questions regarding which library to use.

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.

DEAD LETTERING

A queue that is declared with the x-dead-letter-exchange property sends messages which are either rejected, nacked (negative acknowledged) or expired (with TTL), to the specified dead-letter-exchange. Specifying x-dead-letter-routing-key in the routing key of the message will change it when dead lettered.

TTL

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

PART TWO

BEST PRACTICES FOR HIGH PERFORMANCE

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

Keep queues short

For optimal performance, keep queues as short as possible all the time. Longer queues impose more processing overhead. Queues should always stay around zero (0) for optimal performance.

Set max-length if needed

A feature that is recommended for applications that often receive message spikes is setting a max-length. This keeps the queue short by discarding messages from the head of the queues to keep it no larger than the max-length setting.

Do not use lazy queues

Lazy queues mean that messages are automatically stored to disk, which slows down the throughput. Note that CloudAMQP has lazy queues enabled by default.

Use transit messages

Use transit messages to avoid lowered throughput caused by persistent messages, which are written to disk as soon as they reach the queue.

Use multiple queues and consumers

Queues are single-threaded in RabbitMQ, and one queue can handle up to about 50k messages. Better throughput on a multi-core system is achieved through use of multiple queues and consumers.

The RabbitMQ management interface collects and calculates metrics for every queue in the cluster, which may slow down the server if thousands upon thousands of active queues and consumers are on the system.

Split queues over different cores

Better performance is achieved by splitting queues into different cores, and into different nodes if possible, as queue performance is limited to one CPU core.

Two plugins are recommended that will help systems copy with multiple nodes or a single node cluster with multiple cores; the consistent hash exchange plugin and RabbitMQ sharding plugin.

Disable manual acks and publish confirms

Acknowledgment and publish confirms both have an impact on performance. For the fastest possible throughput, manual acks should be disabled, which will speed up the broker by allowing it to "fire and forget" the message.

Avoid multiple nodes (HA)

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

Enable RabbitMQ HiPE

HiPE increases server throughput, but keep in mind this will be at the cost of increased startup time. Enabling HiPE causes RabbitMQ to be compiled at startup, which affects the throughput by an increase of 20 to 80 percent, according to benchmark tests.

Disable unused plugins

Some plugins might be great, but they also consume a lot of CPU or may use a high amount of RAM. Because of this, they are not recommended for a production server. Disable unused plugins.

PART TWO

BEST PRACTICES FOR HIGH AVAILABILITY

This section is a summary of recommended configurations for high-availability or up-time for the RabbitMQ cluster.

Keep queues short

For optimal performance, keep queues short whenever possible. Longer queues impose more processing overhead, so keep queues around zero (0) for optimal performance.

Enable lazy queues

Lazy queues write messages to disk immediately, which spreads the work out over time instead of risking a performance hit somewhere down the line. The result of a lazy queue is a more predictable, smooth performance curve without sudden drops.

RabbitMQ HA – Two (2) nodes

Enhance availability of data by using replicas means that clients can find messages even through system failures. Two (2) nodes are optimal for high availability, and CloudAMQP locates each node in a cluster in different availability zones (AWS). Additionally, queues are automatically mirrored and 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, the auto-failover mechanism distributes tasks to other nodes in the cluster. RabbitMQ instances include a load balancer, which makes broker 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).

Optional federation use between clouds

Clustering is not recommended between clouds or regions, and therefore there is no plan at CloudAMQP to spread nodes across regions or datacenters. If one cloud region goes down, the CloudAMQP cluster also goes down, but this scenario would be extremely rare. Instead, cluster nodes are spread across availability zones within the same region.

Protect the setup against a region-wide outage by setting up two clusters in different regions and use federation between them. Federation a method in 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, prepare for broker restarts, broker hardware failure, and broker crashes by ensuring that they are on disk. Messages, exchanges, and queues that are not durable and persistent are lost during a broker restart.

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

Do not enable HiPE

HiPE increases server throughput, but keep in mind this will be at the cost of increased startup time. Enabling HiPE causes RabbitMQ to be compiled at startup, which affects the startup time with an increase of one to three minutes. HiPE might affect uptime during a server restart, which affects availability.

Management statistics rate mode

Setting the rate mode of RabbitMQ management statistics to 'detailed' has a serious impact on performance. This setting is not recommended in production.

Limited use of priority queues

Each priority level uses an internal queue on the Erlang VM, which consumes resources. In most uses it is sufficient to have no more than five (5) priority levels.

PART TWO

QUEUE FEDERATION

RabbitMQ supports federated queues, which have several uses; when collecting messages from multiple clusters to a central cluster, when distributing the load of one queue to multiple other clusters, and/or when migrating to another cluster without stopping all producers/consumers.

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 is perfect to use when migrating between two clusters. Consumers and publishers can be moved in any order, and the messages will not be duplicated (which is the case in exchange federation). Instead, messages are transferred to the new cluster when consumers are there. The federated queue will only retrieve messages when it has run out of messages locally, when it has consumers that need messages, and when the upstream queue has "spare" messages that are not being consumed.

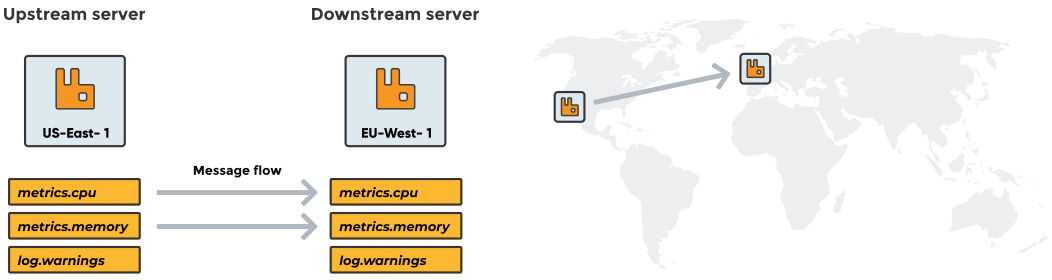


Figure 38 - Upstream and downstream servers.

See Figure 38 for an illustration of the concept of upstream and downstream servers. The upstream server is where messages are initially published, while the downstream server is where the messages are forwarded. Think of the downstream cluster 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 39). 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 server of 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).

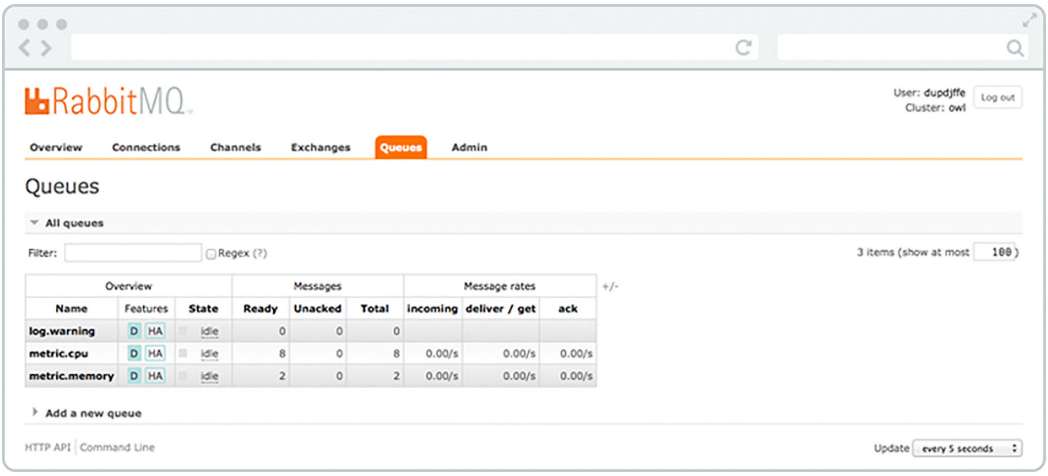


***Figure 39** - Migration of metric.cpu and metric.memory.*

1. **Set up a new cluster.**
   Start by setting up the new cluster; in this case EU-West-1.

2. **Create a policy that matches the queues to federate.**
   A policy is a pattern against which queue names are matched. 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 names begin with "metric".

   Navigate to Admin -> Policies and press Add/update to create the policy (Figure 40). A policy can apply to an upstream set or a single upstream of exchanges and/or queues. In this example, it is applied to all upstream queues with federation-upstream-set set to all.

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

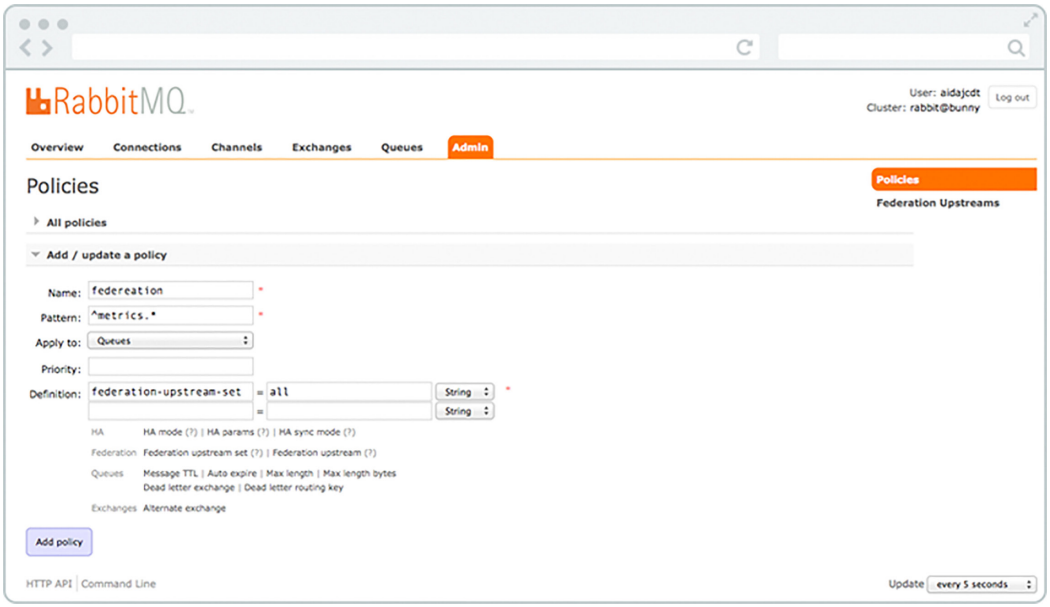


***Figure 40** - Set up the federation to the upstream .*

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

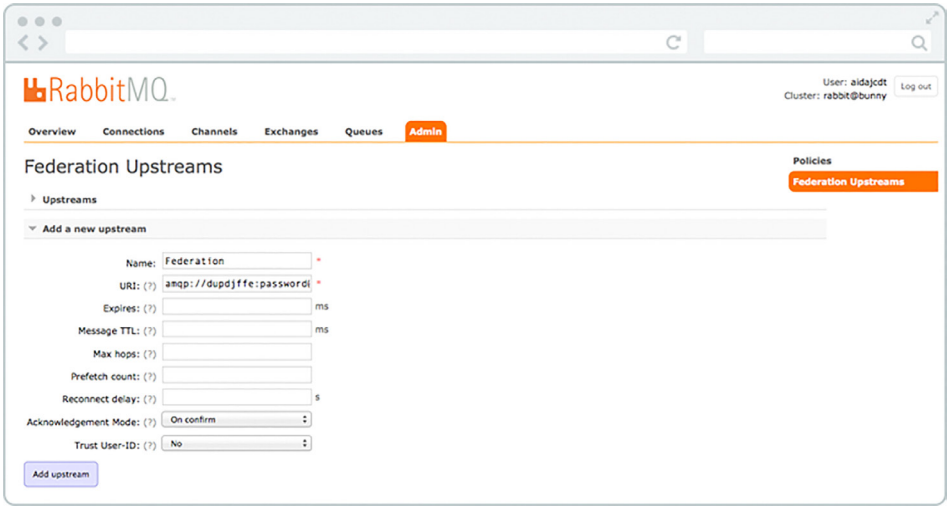


***Figure 41** - Set up the federation to the upstream*

Open the management interface for the downstream server EU-West-1, go to the Admin -> Federation Upstreams screen, and press Add (Figure 41). Fill in all information; the URI should be the URI of the upstream server.

Leave expiry time and TTL blank, which means that the message will stay forever.

4. **Start to move messages.**
   Connect or move the publisher or consumer to the new cluster. If the cluster is migrating, 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.

PART TWO

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.

Message queuing protocols have features in common, so choosing the right one comes down to the use case or scenario. 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 defines the communication between the client and the server and has no impact on the message itself. One protocol can be used when publishing while another can be used to consume. The MQTT protocol, with its minimal design, is perfect for built-in systems, mobile phones, and other memory and bandwidth sensitive applications. While using AMQP for the same task will work, MQTT is a more appropriate choice of protocol for this specific type of scenario.

When creating a CloudAMQP instance, all the common protocols are available by default (exception: web-stomp/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 is 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 for efficient support of a wide variety of messaging applications and communication patterns. AMQP is a more advanced protocol than MQTT, is more reliable, and has better security support. AMQP also has features such as flexible routing, durable and persistent queues, clustering, federation, and high availability queues. The downside is that it is a more verbose protocol, depending solution implementation.

As with other message queuing protocols, the defining features of AMQP are message orientation and queuing. Routing is another feature, which is the process by which an exchange decides which queues to place messages 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. It is designed for built-in systems, mobile phones, and other memory and bandwidth sensitive applications.

MQTT benefits for IoT make a difference for extremely low power devices. MQTT is very code-footprint efficient, as it has a strong focus on using minimal bandwidth. 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 are significant limitations in some scenarios.

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 is a less complex protocol than AMQP, with more similarities to HTTP. STOMP clients can communicate with almost every available STOMP message broker, which provides easy and wide-spread messaging interoperability among many languages, platforms, and brokers. It is for example possible to connect to a STOMP broker using a telnet client.

STOMP does not deal with queues and topics; instead, 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 understood internally). Consumers then SUBSCRIBE to those destinations.

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

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

HTTP

HTTP stands for Hypertext Transfer Protocol, 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 the 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","pay-
load_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:pass@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 p rocessed - 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 permis-

sions, 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 includesocks.min.js and stomp.min.js in your HTML from for example CDNJS:

```
<script src="//cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3.3/stomp.min.js"></script>

// Replace with your hostname
var wss = new WebSocket("wss://blue-horse.rmq.cloudamqp.com/ws/");
var client = Stomp.over(wss);

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

client.debug = onDebug;

// Make sure the user has limited access rights
client.connect("webstomp-user", "webstomp-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 THREE

RABBITMQ USER STORIES

USER STORY

User stories from customers describing their experience using RabbitMQ in production.

A small story about how RabbitMQ is used in the real world can always be of great value. The following are some interesting user stories from various sources that describe how RabbitMQ is used in production.

PART THREE

A MONOLITHIC SYSTEM INTO MICROSERVICES

A growing digital parking service from Sweden is currently breaking down their monolithic application and working toward microservices architecture. (Sept. 2017)

A PORTABLE PARKING METER IN YOUR POCKET

Founded in 2010, Parkster has become one of the fastest growing digital parking services in Sweden. Their vision is to make it quick and easy to pay parking fees with your smartphone, via your Parkster app, with SMS, or with voice. They want to see a world where there is no need 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.

BREAKING UP A TIGHTLY COUPLED MONOLITHIC APPLICATION

Like many other companies, Netflix among them, Parkster started out with a monolithic architecture. They wanted to prove their business model before they went further. A monolithic application means that 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.

In the monolith system, deploying the codebase meant deploying everything at once. One big codebase made it difficult to fix bugs and to add new features. A deep knowledge was also required before attempting a single small code change, as 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 still a work in progress. They are breaking up their software into a collection of small, isolated services, where each service can be deployed and scaled as needed, independently of other services. Their system today has about 15-20 microservices, where the core app is written in Java.

Parkster is already enjoying the change: "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 the codebase has also given the software developers freedom to use whatever technologies made 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 errors in code that could affect the whole application.

Another thing the staff at Parkster likes about their system today is that it can still be operational even if part of the backend processing is delayed or broken. The entire sysem will not break just because one small part is not operating. Breaking up the system into autonomous components has meant that 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 is safe to remove the job from the queue. The message will stay in the queue if anything fails in the receiving application. A message queue provides 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. 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 structure entirely and focus on a new era where the whole system is built upon microservices.

PART THREE

HOW RABBITMQ TRANSFORMED FARMBOT

FarmBot is an open-source robotic hardware kit designed for gardeners, researchers, and educators to interact with agricultural projects in a more efficient way. FarmBot uses physical sensors and actuators that require a bridge between the physical garden and the software layer. In 2017, user demand for real-time response to requests, shorter development time for new features, and a significant improvement in system up-time led Farmbot to RabbitMQ - the world's most popular open-source message broker.

FarmBot's Genesis version can plan, plant, and manage over 30 different crops such as potatoes, peas, squash, and more. Able to manage an area of 2.9 meters × 1.4 meters and a maximum plant height of 0.5 meters, FarmBot Genesis uses manual controls that users move and operate to perform a variety of tasks.

Thanks to the game-like interface that drags and drops plants into a map, FarmBot users immediately grasp the concept of controlling the entire growing season from start to finish. FarmBot tools and peripherals perform tasks such as watering plants, scaring birds away, taking photos of veggies, turning the lights on for a nighttime harvest, or simply impressing friends and neighbors!

On the other side of the coin, FarmBot identifies and eliminates weeds through image analysis via an onboard camera. To practice good conservation efforts, watering is done automatically after taking into account the degree of soil moisture and the weather forecast. There is a web app version of FarmBot to allow users to control garden tasks via smartphone or tablet.

ARCHITECTURE OVERVIEW

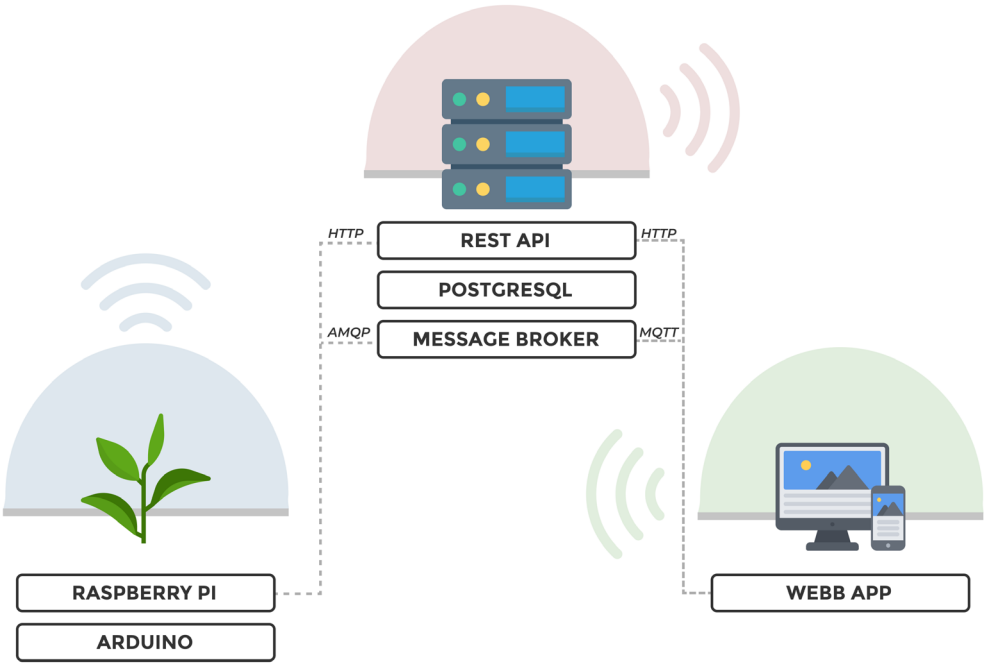


Figure 42 - FarmBot Architectural Overview

FarmBot's architecture consists of three main parts:

1. The vegetable garden, Arduino microcontroller and Raspberry Pi (runs the management software and controls the Arduino CNC)

2. Web server handling requests between the field and the user

3. Web app (on mobile/iPad/browser etc.) that lets the user interact with the system

NOW FOR A DEEP-DIVE

Let’s now take a closer look at all aspects of the system.

Arduino microcontroller (firmware) and Raspberry Pi (FarmBot OS)

The firmware, named Farmduino, is the software responsible for interaction with the real world. It runs on an Arduino microcontroller, written in C++ and executes sensor and actuator commands that allow users to drive the motors, the electro-irrigation valve, and read the probe measurements.

An example of this is when FarmBot turns on the water valve or probes for soil moisture. In this case, it is the device firmware that tells the peripherals directly when and how to operate. Other parts of the system may request access to peripherals by sending commands to the firmware.

A Raspberry Pi running FarmBot OS allows FarmBot to communicate with the web application over WiFi or ethernet to synchronize (download) sequences, regimens, farm designs, events, and more activity, including uploading logs and sensor data and accepting real-time commands. The Raspberry Pi also communicates with the Arduino to send commands and receive sensor and encoder data. It can take photos with a USB or Raspberry Pi camera, and upload the photos to the web application.

The REST API, PostgreSQL, and RabbitMQ

The FarmBot API provides an HTTP based REST API, which handles a number of responsibilities that prevent data loss between reflashes by storing it in a centralized database. This allows users to edit information when the device is offline and allows users to validate and control access to data via authentication and authorization mechanisms. It also sends email notifications (such as password resets and critical errors) to end-users. All other messaging is handled by a message broker - RabbitMQ, a distinctly decoupled sub-system of the Web API. The REST API does not control FarmBot - device control is handled by the Message Broker, CeleryScript and FarmBot JS.

RabbitMQ communicates to the devices in the field by AMQP and acts as a message queue to the backend services. MQTT protocol is used for real-time events to the frontend user interface.

Frontend user interface

The frontend user interface allows users to control and configure FarmBot from any desktop computer, laptop, tablet, or smartphone. Many users end up running their own web server on-premise since it's all open source. FarmBot also provides a publicly accessible server at https://my.farm.bot/ and a staging server at http://staging.farm.bot for end users that are not familiar with Ruby on Rails application development.

WHY RABBITMQ?

Some interactions do not lend themselves well to an HTTP request/response pattern, especially remote procedure calls. A device would be forced to perform long polling, constantly making HTTP requests to the API for any new remote procedure calls. Polling would create tremendous scalability issues for the web app and provide a sub-par real-time experience for users.

For example, emergency stop messages in FarmBot should be received as soon as they are created rather than the system constantly checking the API for messages like these. Other use cases include remote procedure calls and real-time data syncing.

RABBITMQ TO THE RESCUE

In 2017 the Farmbot users got the option to vote for new features to focus on. The vote fell down on auto-sync and better real-time support. This forced the Farmbot community to reevaluate the architecture, and it also made them try something new. Farmbot made the decision to transition their architecture to include RabbitMQ. In a unanimous decision by the FarmBot community, RabbitMQ was the obvious choice, in part because it has:

- Robust support of plugins such as WebSockets and MQTT
- Wide variety of wrapper libraries for Elixir, Ruby, and Typescript
- A good ecosystem, mind share, and ubiquity overall - e.g. it is not an obscure choice

RabbitMQ is now an important component of the FarmBot web API, where it handles various tasks including:

- Passing push notifications between users and devices
- Passing background messages between server background workers
- Uses a set of custom authorization plugins (to prevent unauthorized use)

Because RabbitMQ is a real-time message broker, there is no need to check for new messages. Messages, such as a user clicking the "move" button on the user interface, are sent back and forth between client, device, and server without initiating requests.

In many ways, the message broker acts as a machine-to-machine chat application. Any software package, whether it be the REST API, FarmBot OS or third-party firmware, can send a message to any other entity that is currently connected to the message broker, provided it has the correct authorization.

THE REST IS SUCCESS

Today, FarmBot is a happy, satisfied customer of the largest hosting provider of RabbitMQ, CloudAMQP. “The interactions with customer service have been quick and pleasant, and we've seen great uptime stats,“ said Rick Carlino, Lead Software Developer at FarmBot.

He continues, “We really don't have time to manage our own services on AWS and would much rather pay a premium to know that someone else is taking care of the problem if there is one.”

Choosing the right technology and the right vendor often focuses on the best selection to optimize developer time. To address this, CloudAMQP provides support around the clock including built-in alarms that can be set up to detect issues with the server before it affects the business.

HEROKU - CLOUD APPLICATION PLATFORM

The same goes for the decision to host the production servers on Heroku. Even though there would be cost savings in hosting production a traditional AWS/Kubernetes setup, it's simply too much of a risk for a small company to save only a couple hundred dollars a month. FarmBot’s Carlino added, “Every hour I spend thinking about infra is an hour that I am not spending on feature development.”

THE CLOUDAMQP ADVANTAGE

User demand for real-time response to requests, shorter development time for new features, and a significant improvement in system up-time all led FarmBot to choose RabbitMQ and CloudAMQP.

As the leading hosting provider of RabbitMQ, CloudAMQP was able to help Farmbot to quickly format their old architecture into a message queue based system, transforming the agri-tech app into a farming powerhouse.

As FarmBot leads the way in the innovation of how food is grown, CloudAMQP is leading the RabbitMQ revolution through user success stories like this one. Let us know if it’s time to create your own successful message queue architecture? Get in touch with our friendly team of professionals today to determine the right RabbitMQ plan for your future growth.

PART THREE

MICROSERVICE ARCHITECTURE BUILT UPON RABBITMQ

We at CloudAMQP rely on RabbitMQ in our everyday life; in fact, a huge number of our events in the production environment pass 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 is built from scratch on small, independent, manageable services that communicate with each other - microservices. These microservices are all highly decoupled and focused on their specific task. This chapter gives an overview and a deeper insight into the automated process behind CloudAMQP, describing some of our microservices and the use of RabbitMQ as a message broker communicating between them.

BACKGROUND OF CLOUDAMQP

A few years ago, Carl Hörberg, the CEO of 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. A message can be sent 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. The pub-sub (publish-subscribe) pattern is followed, as is the retry upon failure process.

Creating a CloudAMQP instance provides the option to choose a plan and how many nodes to have. The cluster will behave a bit different depending on the cluster setup. The option to create your instance in a dedicated VPC and select RabbitMQ version is also available.

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, or Azure marketplace, just to mention a few.

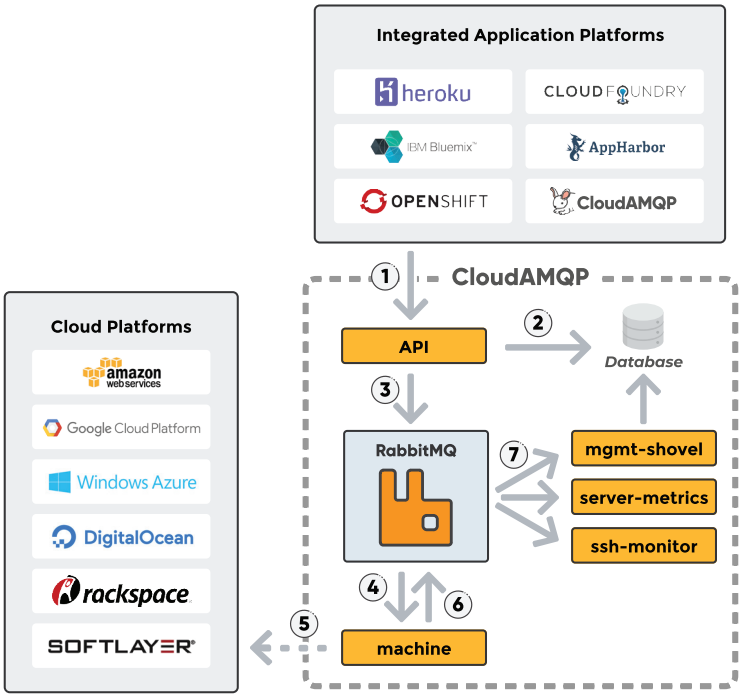


***Figure 43** - The automated process behind CloudAMQP*

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 shown in Figure 43 above. CloudAMQP-API handles the request, saves information into a database (2), and finally, sends a account.create-message to one of our RabbitMQ-clusters (3).

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

CloudAMQP-machine triggers multiple scripts and processes. First, it creates the new server(s) in the chosen datacenter via an HTTP request (5). Different underlying instance types are used, 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. Then the message is sent on the topic exchange (6). The great thing about the topic exchange is that multiple services can subscribe to the event. There are several services listening to account.created-messages (7), all of which will set up a connection to the new server. Here are three examples of services receiving a message and working toward new servers.

Figure 44 - The microservice CloudAMQP-put-metrics

CloudAMQP-server-metrics Continuously gathers 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 Monitoring of the many processes required on all servers.

CloudAMQP has many other services communicating with those described above and with the new server(s) created for the client.

CloudAMQP-server-metrics

CloudAMQP-server-metrics collects metrics (CPU/memory/disk data) for all running servers. The collected metric data 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

Different services subscribe to server metrics data. One of these services is called CloudAMQP-alarm. CloudAMQP-alarm checks the server metrics from RabbitMQ against the alarm thresholds for the given server, and notifies the owner of the server if needed. Users are able to enable/disable alarms such as CPU or memory, for example, as they see fit.

CloudAMQP-put-metrics

CloudAMQP integrates the monitoring tools DataDog, Logentries, AWS Cloudwatch, Google Cloud Stackdriver Logging, and Librato, that are user enabled. 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.

This section described a small part of the CloudAMQP service, which includes around 100 microservices overall, all communicating via CloudAMQP and RabbitMQ.

PART THREE

EVENT-BASED COMMUNICATION

Softonic, a software and app discovery portal, is accessed by over 100 million users per month and delivers more than 2 million downloads per day, with a constant flow of events and commands between services. As the world's largest software and app discovery destination, Softonic is also one of the world's most highly-trafficked websites.

Even without realizing it, you have probably landed on their website when downloading software or an application. Over 100 million users per month rely on Softonic as an app guide that assists with the discovery of the best applications for any device that also serves up reviews, news, articles, and free downloads.

CloudAMQP provides hosted RabbitMQ clusters in the biggest data centers around the world and Softonic is one of our valued customers. The CloudAMPQ team met up with Riccardo Piccoli, a developer at Softonic, at the RabbitMQ Summit 2018 in London where he kindly shared Softonic's customer story with us.

This article is broken down into two parts; the first part is an overview of the system, which shows a simple RabbitMQ use cases of an event-based architecture. The second part is a deep-dive into the internal architecture in Softonic and the plugins used by the company along with examples of events they are sending.

A SIMPLE RABBITMQ USE CASE

Users are able to upload files to Softonic. The system first scans the uploaded file for viruses and collects basic information. After the information is collected, the file is ready for distribution to other users.

The new binary data is first held within a dedicated service, and a notification about the upload is sent to an event bus. Other services collect this information which is eventually added to the Softonic website. In this case, the user gets notified immediately after the upload has succeeded and a scanning event is simply placed on an event-bus for other services to handle.

The message queue, in this section called an event-bus, allows web servers to respond to requests quickly instead of being forced to perform a resource-heavy process on the spot, which could cause user wait-time issues. Virus scanning is an example of a resource-heavy process. The virus scanning application takes a message from the event bus, such as a "ScanFile" command, and starts processing. At the same time, other users are able to upload new files to Softonic and processing tasks are able to join the queue. The event "FileScanned" is added back to the event bus, once the resource-consuming application has handled the event.

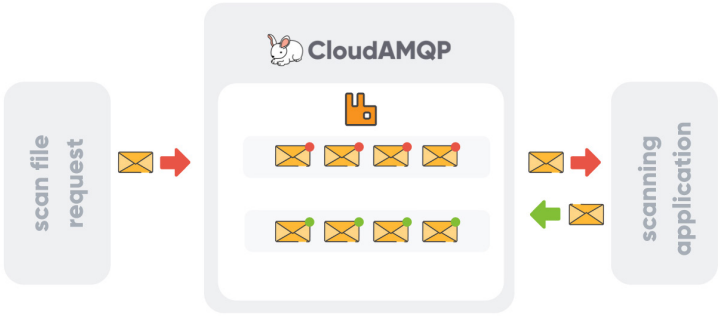


Figure 45 - Scan file request

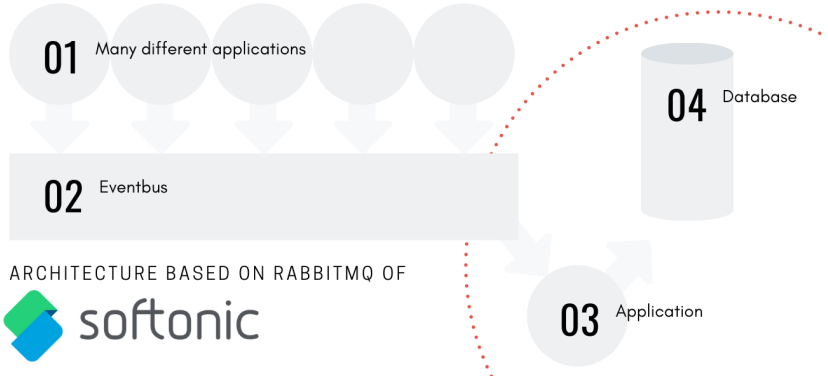


Figure 46 - Softonic RabbitMQ architecture

Architecture like this creates two simple services and low coupling between the sender and the receiver. Users can still upload files, even if the scanning application is busy or is under maintenance.

1. Different events or commands are published to the event bus, e.g., a "ScanFile" command.

2. Softonic uses RabbitMQ as an event bus, wherein events or commands are simply added to the queue.

3. The resource-consuming application retrieves the event and starts to process it, while some data is stored to the database and more events can be published back to another event queue (more about this in "Internal Structure of RabbtiMQ").

4. The resource-consuming application is able to store a great deal of information in a database (MySQL).

When a microservice receives an event, it can update its own status, which leads to more events being published, which is the case here.

THE INTERNAL STRUCTURE OF RABBITMQ

It's time for a deep-dive into the internal architecture of RabbitMQ and into the Softonic application. Softonic is using the Consistent Hash Exchange Plugin and RabbitMQ Sharding.

Image description external usage: Softonic services are built upon Node.js and PHP and communicate with the RabbitMQ event bus, from which information from the services are transferred from a PHP application to a MySQL event store.

Image description internal usage: Information from the first application retrieves data from the MySQL Event Share and pushes it through consistent hash exchanges in two internal RabbitMQ event buses using sharded queues. From there, the information reaches the orchestration layer and an elasticsearch cluster, where it becomes visible for users.

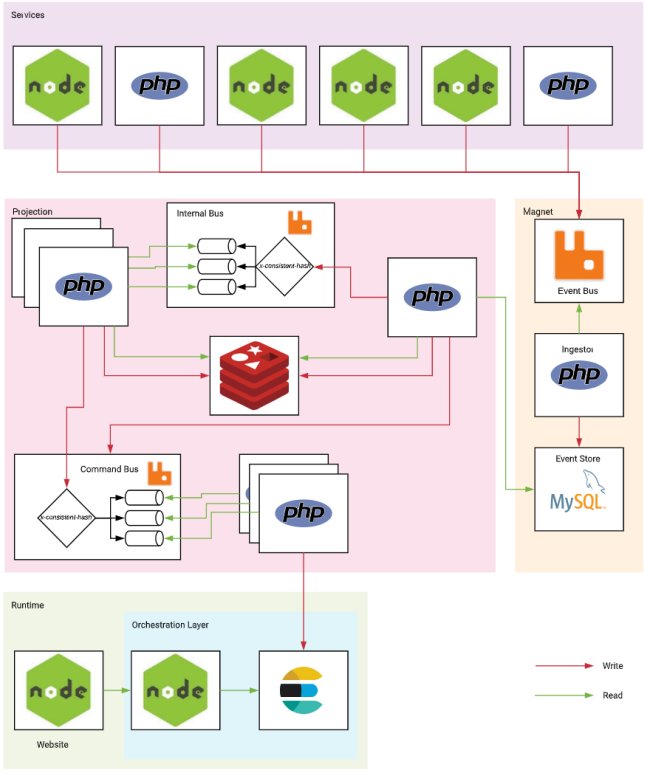


***Figure 47** - Softonic internal architecture*

The consistent hash exchange plugin and RabbitMQ sharding

The consistent hash exchange plugin has the task of load balancing 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 distributes the messages between queues that have a binding to that exchange.

The RabbitMQ Sharding plugin partitions queues automatically. Once you define 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 could be many queues running in the background. .

An example sequence of Softonic events and commands

Below is an example of events and commands sent via RabbitMQ by Softonic using the consistent hash exchange plugin. Events 1 and 2 end up in the same queue (with the order preserved) while event 3 may or may not end up in the same queue. Data is sharded, and processed with consistent hashing F(id_program) in order to preserve order by the program.

- Event 0: Create category “antivirus” (name: “antivirus”)
- Event 1: Create program A (name “foobar”, category “antivirus”, developer “softonic”)
- Event 2: Create a review for program A
- Event 3: Create program B (name “foo”, category “antivirus”, developer “84codes”)
- Event 4: Update category “antivirus” name to “Antivirus”

In this example, event 0 and event 4 need to be processed synchronously, while events 1, 2, and 3 can be processed asynchronously. Event 0 will be processed immediately and events 1,2, and 3 will be re-published to the queue so that other sharded consumers can process them.

CloudAMQP - Message queuing as a service

Softonic did run RabbitMQ in-house before moving to the cloud. The biggest reason for choosing CloudAMQP as a provider was for simplicity of installing RabbitMQ without the hassle of maintaining a RabbitMQ cluster.

CloudAMQP offers many different plans designed for different uses, including a free plan with Little Lemur.

The CloudAMQP team is grateful for the information from Softonic, an impressive example of microservice architecture. A special thanks to Riccardo for your time at the

RabbitMQ Summit 2018, hope to see you again at the next event. We wish Softonic the best of luck with their continued success!

> "The biggest reason for choosing CloudAMQP as a provider was for simplicity of installing RabbitMQ without the hassle of maintaining a RabbitMQ cluster."
>
> *- Riccardo Piccoli, Softonic*