RabbitMQ

Michele Beretta

michele.beretta@unibg.it

What is RabbitMQ?

RabbitMQ is a powerful, enterprise grade open source messaging and streaming broker that enables efficient, reliable and versatile communication for applications — perfect for distributed microservices, real-time data, and IoT.

Released under the *Mozilla Public License 2.0*, it's built in Erlang and uses the Open Telecom Platform (OTP).

So, what is a message broker?

Wikipedia says: a **message broker** is an architectural pattern for message validation, transformation, and routing. It mediates communication among applications, minimising the mutual awareness that applications should have of each other in order to be able to exchange messages, effectively implementing decoupling.

Basic building blocks

What we want to obtain, at the simplest level, is this

That is:

- A **producer** P that sends messages
- A **client** C that receives messages
- And a **queue**, here called `hello`, to store them temporarily

What the heck is a queue?

A queue is like a **post box** in RabbitMQ. Messages flow in your application, but are stored in the queue. A queue is only bound by the host's memory & disk limits: it's essentially a (very very) big array.

Show me the code

This is the *producer*

```python
#!/usr/bin/env python

import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.queue_declare(queue='hello')

channel.basic_publish(exchange='', routing_key='hello', body='Execute order 66 please')
print('(i) Order sent')

connection.close()
```

Show me the code

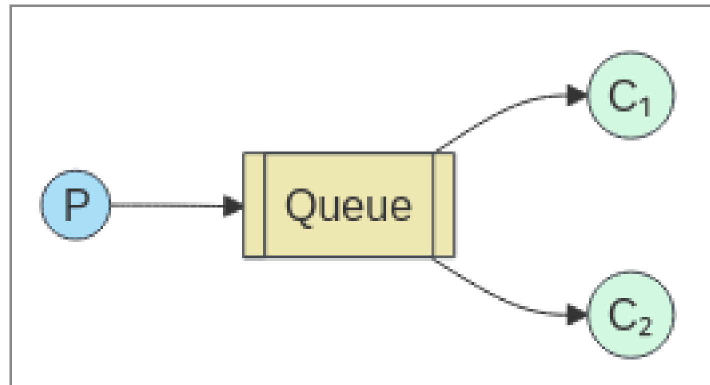
And this the *client*

```python
#!/usr/bin/env python

import pika

def handle_message(channel, method, props, body):
    print('(i) Received a message!')
    print(body)

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.queue_declare(queue='hello')

channel.basic_consume(queue='hello', on_message_callback=handle_message, auto_ack=True)
channel.start_consuming()
```

All good, but what if I have a lot of messages?

For that, we can use a *Work Queue*.

The main idea behind *Work Queues* (or Task Queues) is to avoid doing a resource-intensive task immediately and having to wait for it to complete.



We encapsulate a task as a message and send it to the queue. A worker process running in the background will pop the tasks and eventually execute the job. When you run many workers the tasks will be shared between them.

Show me the code

Essentially the same as before. The *producer*:

```python
#!/usr/bin/env python

import sys
import pika

message = ' '.join(sys.argv[1:]) or 'Just a default message'

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.queue_declare(queue='tasks')

channel.basic_publish(exchange='', routing_key='tasks', body=message)
print('(i) Message sent')

connection.close()
```

Show me the code

And the *client*:

```python
#!/usr/bin/env python

import pika
import time

def handle_message(channel, method, props, body):
    print(f'Received: {body.decode()}')
    time.sleep(5) # simulate some hard work
    channel.basic_ack(delivery_tag=method.delivery_tag)

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))

channel = connection.channel()
channel.queue_declare(queue='tasks')
channel.basic_consume(queue='tasks', on_message_callback=handle_message)
channel.start_consuming()
```

What is with the `ack`?

Not always doing something is instantaneous.

So, what happens if a consumer starts a long task and is **killed** before it could end? With the previous code RabbitMQ would schedule the message for **deletion** as soon as it was delivered.

Hence, we introduce **message acknowledgments**: they simply allow the consumer to tell RabbitMQ "hey, I got the message, you can delete it".

By sending the ACK at work done, if a consumer dies, the message is never acknowledged until some worker finishes the job!

Ok but what happens if RabbitMQ itself crashes?

We would lose the messages, plain and simple. BUT, we can do something. We can

- Make the queue **durable**, so that the queue itself survives RabbitMQ dying
- And make the messages **persistent**

How to do this is very simple, just a couple of options in the library (hence left as an exercise for the reader).

The next step: full on PubSub with routing

What if not all clients should read all messages?

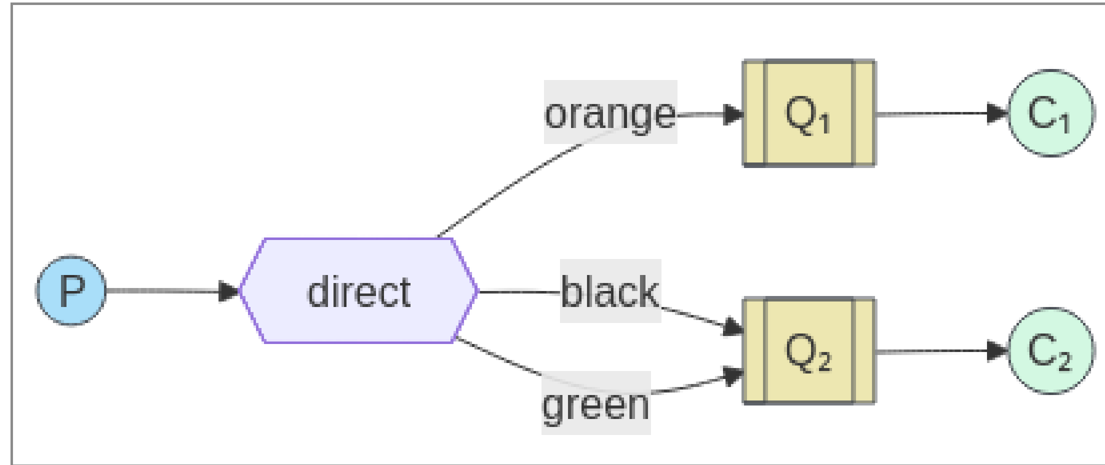
Or maybe one client is interested only in a specific category of messages?

We have to unmask a little white lie: producers *don't* send to queues. They send to *exchanges*, and then these things send messages to queues.

There are a few exchange types: `direct`, `topic`, `headers`, and `fanout`. Here, `fanout` is your usual `broadcast`, while `direct` allows for the filtering of messages.

Moreover, previously we had to have queue names: now, we can let the server decide and get a random queue for every client.

An example architecture



Here we have one producers that sends messages to one `direct` exchange.

Then

- C_1 is only interested in `orange` messages
- C_2 only wants to see `black` or `green` messages

In this case, they create a random queue when attaching to the exchange, and the exchange knows what to send to each queue.

Show me the code: logs

We can produce three type of logs: `info`, `warning`, and `error`

```
#!/usr/bin/env python
import pika
import sys

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='direct_logs', exchange_type='direct')

severity = sys.argv[1] if len(sys.argv) > 1 else 'info'
message = ' '.join(sys.argv[2:]) or 'hello there'
channel.basic_publish(exchange='direct_logs', routing_key=severity, body=message)

print(f'(i) Sent {severity}:{message}')
connection.close()
```

Show me the code

```
#!/usr/bin/env python

# Setup code omitted (it's the same as other examples)
channel.exchange_declare(exchange='direct_logs', exchange_type='direct')
result = channel.queue_declare(queue='', exclusive=True)

for severity in sys.argv[1:]:
    channel.queue_bind(exchange='direct_logs', queue=result.method.queue,
routing_key=severity)

print('(i) Waiting for logs. To exit press CTRL+C')

channel.basic_consume(queue=result.method.queue, on_message_callback=handle_message,
auto_ack=True)
channel.start_consuming()
```