

1 "Hello World!"

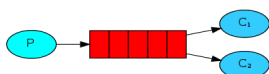
The simplest thing that does something



Python | Java

2 Work queues

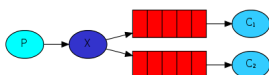
Distributing tasks among workers



Python | Java

3 Publish/Subscribe

Sending messages to many consumers at once



Python | Java

4 Routing

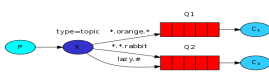
Receiving messages selectively



Python | Java

5 Topics

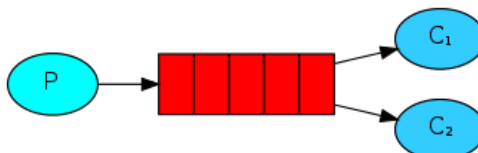
Receiving messages based on a pattern



Python | Java

Work Queues

(using the pika 0.9.8 Python client)



Where to get help

If you're having trouble going through this tutorial you can **contact us** through the discussion list or directly.

In the **first tutorial** we wrote programs to send and receive messages from a named queue. In this one we'll create a *Work Queue* that will be used to distribute time-consuming tasks among multiple workers.

The main idea behind Work Queues (aka: *Task Queues*) is to avoid doing a resource-intensive task immediately and having to wait for it to complete. Instead we schedule the task to be done later. 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.

This concept is especially useful in web applications where it's impossible to handle a complex task during a short HTTP request window.

Preparation

In the previous part of this tutorial we sent a message containing "Hello World!". Now we'll be sending strings that stand for complex tasks. We don't have a real-world task, like images to be resized or pdf files to be rendered, so let's fake it by just pretending we're busy - by using the `time.sleep()` function. We'll take the number of dots in the string as its complexity; every dot will account for one second of "work". For example, a fake task described by `hello...` will take three seconds.

We will slightly modify the `send.py` code from our previous example, to allow arbitrary messages to be sent from the command line. This program will schedule tasks to our work queue, so let's name it `new_task.py`:

```
import sys

message = ' '.join(sys.argv[1:]) or "Hello World!"
channel.basic_publish(exchange='',
                    routing_key='hello',
                    body=message)

print " [x] Sent %r" % (message,)
```

Our old `receive.py` script also requires some changes: it needs to fake a second of work for every dot in the message body. It will pop messages from the queue and perform the task, so let's call it `worker.py`:

```
import time

def callback(ch, method, properties, body):
    print " [x] Received %r" % (body,)
    time.sleep( body.count('.') )
    print " [x] Done"
```

Round-robin dispatching

One of the advantages of using a Task Queue is the ability to easily parallelise work. If we are building up a backlog of work, we can just add more workers and that way, scale easily.

First, let's try to run two `worker.py` scripts at the same time. They will both get messages from the queue, but how exactly? Let's see.

You need three consoles open. Two will run the `worker.py` script. These consoles will be our two consumers - C1 and C2.

```
shell1$ python worker.py
[*] Waiting for messages. To exit press CTRL+C
```

6 RPC

Remote procedure call implementation



Python | Java

```
shell2$ python worker.py
[*] Waiting for messages. To exit press CTRL+C
```

In the third one we'll publish new tasks. Once you've started the consumers you can publish a few messages:

```
shell3$ python new_task.py First message.
shell3$ python new_task.py Second message..
shell3$ python new_task.py Third message...
shell3$ python new_task.py Fourth message....
shell3$ python new_task.py Fifth message.....
```

Let's see what is delivered to our workers:

```
shell1$ python worker.py
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'First message.'
[x] Received 'Third message...'
[x] Received 'Fifth message.....'
```

```
shell2$ python worker.py
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'Second message..'
[x] Received 'Fourth message....'
```

By default, RabbitMQ will send each message to the next consumer, in sequence. On average every consumer will get the same number of messages. This way of distributing messages is called round-robin. Try this out with three or more workers.

Message acknowledgment

Doing a task can take a few seconds. You may wonder what happens if one of the consumers starts a long task and dies with it only partly done. With our current code once RabbitMQ delivers message to the customer it immediately removes it from memory. In this case, if you kill a worker we will lose the message it was just processing. We'll also lose all the messages that were dispatched to this particular worker but were not yet handled.

But we don't want to lose any tasks. If a worker dies, we'd like the task to be delivered to another worker.

In order to make sure a message is never lost, RabbitMQ supports message *acknowledgments*. An *ack*(nowledgement) is sent back from the consumer to tell RabbitMQ that a particular message had been received, processed and that RabbitMQ is free to delete it.

If consumer dies without sending an *ack*, RabbitMQ will understand that a message wasn't processed fully and will redeliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die.

There aren't any message timeouts; RabbitMQ will redeliver the message only when the worker connection dies. It's fine even if processing a message takes a very, very long time.

Message acknowledgments are turned on by default. In previous examples we explicitly turned them off via the `no_ack=True` flag. It's time to remove this flag and send a proper acknowledgment from the worker, once we're done with a task.

```
def callback(ch, method, properties, body):
    print "[x] Received %r" % (body,)
    time.sleep( body.count('.') )
    print "[x] Done"
    ch.basic_ack(delivery_tag = method.delivery_tag)

channel.basic_consume(callback,
                      queue='hello')
```

Using this code we can be sure that even if you kill a worker using CTRL+C while it was processing a message, nothing will be lost. Soon after the worker dies all unacknowledged messages will be redelivered.

Forgotten acknowledgment

It's a common mistake to miss the `basic_ack`. It's an easy error, but the consequences are serious. Messages will be redelivered when your client quits (which may look like random redelivery), but RabbitMQ will eat more and more memory as it won't be able to release any unacked messages.

In order to debug this kind of mistake you can use `rabbitmqctl` to print the `messages_unacknowledged` field:

```
$ sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged
Listing queues ...
hello      0          0
...done.
```

Message durability

We have learned how to make sure that even if the consumer dies, the task isn't lost. But our tasks will still be lost if RabbitMQ server stops.

When RabbitMQ quits or crashes it will forget the queues and messages unless you tell it not to. Two things are required to make sure that messages aren't lost: we need to mark both the queue and messages as durable.

First, we need to make sure that RabbitMQ will never lose our queue. In order to do so, we need to declare it as *durable*:

```
channel.queue_declare(queue='hello', durable=True)
```

Although this command is correct by itself, it won't work in our setup. That's because we've already defined a queue called `hello` which is not durable. RabbitMQ doesn't allow you to redefine an existing queue with different parameters and will return an error to any program that tries to do that. But there is a quick workaround - let's declare a queue with different name, for example `task_queue`:

```
channel.queue_declare(queue='task_queue', durable=True)
```

This `queue_declare` change needs to be applied to both the producer and consumer code.

At that point we're sure that the `task_queue` queue won't be lost even if RabbitMQ restarts. Now we need to mark our messages as persistent - by supplying a `delivery_mode` property with a value 2.

```
channel.basic_publish(exchange='',
                    routing_key="task_queue",
                    body=message,
                    properties=pika.BasicProperties(
                        delivery_mode = 2, # make message persistent
                    ))
```

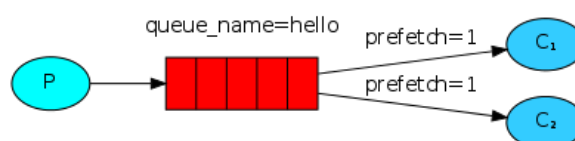
Note on message persistence

Marking messages as persistent doesn't fully guarantee that a message won't be lost. Although it tells RabbitMQ to save message to the disk, there is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet. Also, RabbitMQ doesn't do `fsync(2)` for every message -- it may be just saved to cache and not really written to the disk. The persistence guarantees aren't strong, but it's more than enough for our simple task queue. If you need a stronger guarantee you can wrap the publishing code in a *transaction*.

Fair dispatch

You might have noticed that the dispatching still doesn't work exactly as we want. For example in a situation with two workers, when all odd messages are heavy and even messages are light, one worker will be constantly busy and the other one will do hardly any work. Well, RabbitMQ doesn't know anything about that and will still dispatch messages evenly.

This happens because RabbitMQ just dispatches a message when the message enters the queue. It doesn't look at the number of unacknowledged messages for a consumer. It just blindly dispatches every n-th message to the n-th consumer.



In order to defeat that we can use the `basic.qos` method with the `prefetch_count=1` setting. This tells RabbitMQ not to give more than one message to a worker at a time. Or, in other words, don't dispatch a new message to a worker until it has processed and acknowledged the previous one. Instead, it will dispatch it to the next worker that is not still busy.

```
channel.basic_qos(prefetch_count=1)
```

Note about queue size

If all the workers are busy, your queue can fill up. You will want to keep an eye on that, and maybe add more workers, or have some other strategy.

Putting it all together

Final code of our `new_task.py` script:

```
1  #!/usr/bin/env python
2  import pika
3  import sys
4
5  connection = pika.BlockingConnection(pika.ConnectionParameters(
6      host='localhost'))
7  channel = connection.channel()
8
9  channel.queue_declare(queue='task_queue', durable=True)
10
11 message = ' '.join(sys.argv[1:]) or "Hello World!"
12 channel.basic_publish(exchange='',
13                       routing_key='task_queue',
14                       body=message,
15                       properties=pika.BasicProperties(
16                           delivery_mode = 2, # make message persistent
17                       ))
18 print " [x] Sent %r" % (message,)
19 connection.close()
```

(new_task.py source)

And our worker:

```
1  #!/usr/bin/env python
2  import pika
3  import time
4
5  connection = pika.BlockingConnection(pika.ConnectionParameters(
6      host='localhost'))
7  channel = connection.channel()
8
9  channel.queue_declare(queue='task_queue', durable=True)
10 print " [*] Waiting for messages. To exit press CTRL+C"
11
12 def callback(ch, method, properties, body):
13     print " [x] Received %r" % (body,)
14     time.sleep( body.count('.') )
15     print " [x] Done"
16     ch.basic_ack(delivery_tag = method.delivery_tag)
17
18 channel.basic_qos(prefetch_count=1)
19 channel.basic_consume(callback,
20                       queue='task_queue')
21
22 channel.start_consuming()
```

(worker.py source)

Using message acknowledgments and `prefetch_count` you can set up a work queue. The durability options let the tasks survive even if RabbitMQ is restarted.

Now we can move on to **tutorial 3** and learn how to deliver the same message to many consumers.

