RabbitMQ Tutorials

<http://www.rabbitmq.com/tutorials/tutorial-one-python.html>

**"Hello World!"**

**Introduction**

RabbitMQ is a message broker: it accepts and forwards messages. You can think about it as a post office: when you put the mail that you want posting in a post box, you can be sure that Mr. Postman will eventually deliver the mail to your recipient. In this analogy, RabbitMQ is a post box, a post office and a postman.

RabbitMQ是一个消息代理。它的核心思想非常简单：接收并转发消息。你可以把它想象成一个邮局：当你把邮件丢进邮箱时，你非常确定邮递员先生会把它送到收件人手中。在这个比喻中，RabbitMQ就是邮箱、邮局和邮递员。

The major difference between RabbitMQ and the post office is that it doesn't deal with paper, instead it accepts, stores and forwards binary blobs of data ‒ messages.

RabbitMQ和邮局的主要区别是它处理的不是纸张。它接收、存储并转发二进制数据块，也就是message，消息。

RabbitMQ, and messaging in general, uses some jargon.

通常来讲，RabbitMQ和消息传递中会用到一些术语：

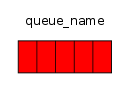
Producing means nothing more than sending. A program that sends messages is a producer :

* producing的意思是发送。一个发送消息的程序叫做producer。图中我们用一个“P”来表示它：

IMG_256

A queue is the name for a post box which lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can only be stored inside a queue. A queue is only bound by the host's memory & disk limits, it's essentially a large message buffer. Many producers can send messages that go to one queue, and many consumers can try to receive data from one queue. This is how we represent a queue:

* 一个queue，即队列，相当于一个邮箱，它由RabbitMQ管理。尽管消息会在你的应用和RabbitMQ之间流过，但他们只被保存在队列中。队列没有边界限制，你想存多少消息就能存多少——它本质上是一个无限制的缓冲区。一个队列可以接收多个producer的消息，也可以被多个consumer读取。下图是一个队列，上边的字代表它的名字：



Consuming has a similar meaning to receiving. A consumer is a program that mostly waits to receive messages:

* consuming的意思类似于接收。一个等待接收消息的程序叫做consumer。在图中我们用一个“C”来表示它。

IMG_256

Note that the producer, consumer, and broker do not have to reside on the same host; indeed in most applications they don't.

要注意，producer、consumer和消息代理不需要生活在同一台机器上，事实上大多数应用中它们会分开住。

**Hello World!**

(using the pika 0.10.0 Python client)

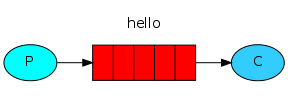
In this part of the tutorial we'll write two small programs in Python; a producer (sender) that sends a single message, and a consumer (receiver) that receives messages and prints them out. It's a "Hello World" of messaging.

在这部分教程中，我们会写两个Python程序。一个发送一条消息的producer和一个接收消息并打印出来的consumer。因为只是刚刚起步，我们会忽略一些细节，只把精力集中在简单的事情上。消息的内容是“Hello World”。

In the diagram below, "P" is our producer and "C" is our consumer. The box in the middle is a queue - a message buffer that RabbitMQ keeps on behalf of the consumer.

下图中，“P”是我们的producer，“C”是我们的consumer。而中间的方形盒子是我们的队列——RabbitMQ代表消费者保留的消息缓冲区

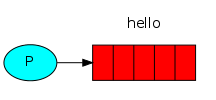
Our overall design will look like:



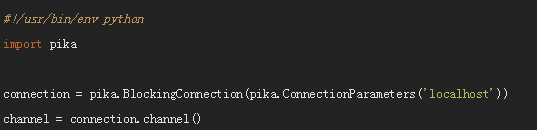
RabbitMQ libraries

RabbitMQ speaks AMQP 0.9.1, which is an open, general-purpose protocol for messaging. There are a number of clients for RabbitMQ in many different languages. In this tutorial series we're going to use Pika, which is the Python client recommended by the RabbitMQ team. To install it you can use the pip package management tool.

**Sending**



Our first program send.py will send a single message to the queue. The first thing we need to do is to establish a connection with RabbitMQ server.



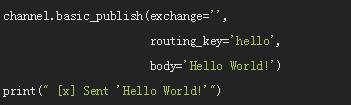
We're connected now, to a broker on the local machine - hence the localhost. If we wanted to connect to a broker on a different machine we'd simply specify its name or IP address here.

Next, before sending we need to make sure the recipient queue exists. If we send a message to non-existing location, RabbitMQ will just drop the message. Let's create a hello queue to which the message will be delivered:



At this point we're ready to send a message. Our first message will just contain a string Hello World! and we want to send it to our hello queue.

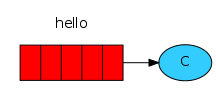
In RabbitMQ a message can never be sent directly to the queue, it always needs to go through an exchange. But let's not get dragged down by the details ‒ you can read more about exchanges in the third part of this tutorial. All we need to know now is how to use a default exchange identified by an empty string. This exchange is special ‒ it allows us to specify exactly to which queue the message should go. The queue name needs to be specified in the routing\_key parameter:



Before exiting the program we need to make sure the network buffers were flushed and our message was actually delivered to RabbitMQ. We can do it by gently closing the connection.



**Receiving**



Our second program receive.py will receive messages from the queue and print them on the screen.

Again, first we need to connect to RabbitMQ server. The code responsible for connecting to Rabbit is the same as previously.

The next step, just like before, is to make sure that the queue exists. Creating a queue using queue\_declare is idempotent ‒ we can run the command as many times as we like, and only one will be created.



You may ask why we declare the queue again ‒ we have already declared it in our previous code. We could avoid that if we were sure that the queue already exists. For example if send.py program was run before. But we're not yet sure which program to run first. In such cases it's a good practice to repeat declaring the queue in both programs.

* Listing queues

You may wish to see what queues RabbitMQ has and how many messages are in them. You can do it (as a privileged user) using the rabbitmqctl tool:



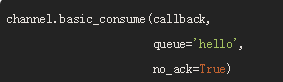
On Windows, omit the sudo:



Receiving messages from the queue is more complex. It works by subscribing a callback function to a queue. Whenever we receive a message, this callback function is called by the Pika library. In our case this function will print on the screen the contents of the message.



Next, we need to tell RabbitMQ that this particular callback function should receive messages from our hello queue:



For that command to succeed we must be sure that a queue which we want to subscribe to exists. Fortunately we're confident about that ‒ we've created a queue above ‒ using queue\_declare.

The no\_ack parameter will be described later on.

And finally, we enter a never-ending loop that waits for data and runs callbacks whenever necessary.



Hurray! We were able to send our first message through RabbitMQ. As you might have noticed, the receive.py program doesn't exit. It will stay ready to receive further messages, and may be interrupted with Ctrl-C.

Try to run send.py again in a new terminal.

We've learned how to send and receive a message from a named queue. It's time to move on to part 2 and build a simple work queue.

**Work Queues**

**Prerequisites**

As with other Python tutorials, we will use the Pika RabbitMQ client version 0.11.0.



**What This Tutorial Focuses On**

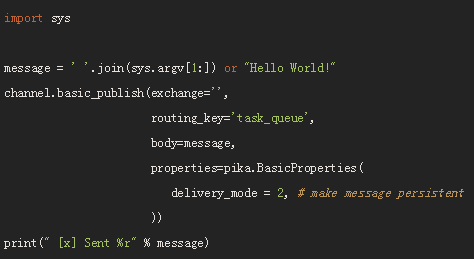
In the first tutorial we wrote programs to send and receive messages from a named queue. In this one we'll create a Work Queue that will be used to distribute time-consuming tasks among multiple workers.

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.

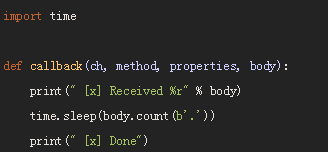
This concept is especially useful in web applications where it's impossible to handle a complex task during a short HTTP request window.

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:



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:

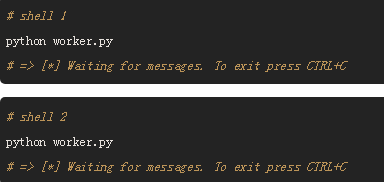


**Round-robin dispatching**

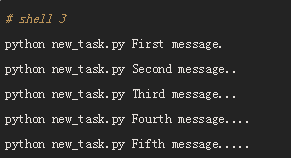
One of the advantages of using a Task Queue is the ability to easily parallelise work. If we are building up a backlog of work, we can just add more workers and that way, scale easily.

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.

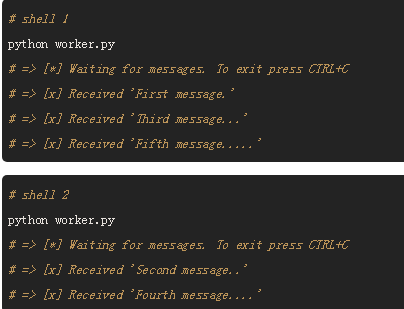
You need three consoles open. Two will run the worker.py script. These consoles will be our two consumers - C1 and C2.



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



Let's see what is delivered to our workers:



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 marks it for deletion. 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.

（这里我自己试验过，我发送6条消息给2个worker，正常每个worker消费3条消息，如果我发完六条消息后，2个worker还在处理前2条消息，然后我把worker 1杀掉，worker 2继续运行，结果worker 2只处理自己的3条消息。所以消息是即时分发到worker的，如果worker死掉，那么分发到它的消息也会消失掉没有处理。）

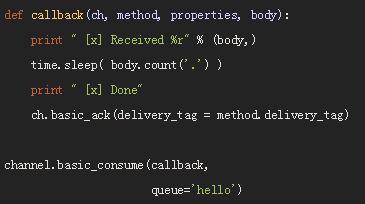
But we don't want to lose any tasks. If a worker dies, we'd like the task to be delivered to another worker.

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

If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it. If there are other consumers online at the same time, it will then quickly 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 when the consumer dies. It's fine even if processing a message takes a very, very long time.

Manual 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.



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:



**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:

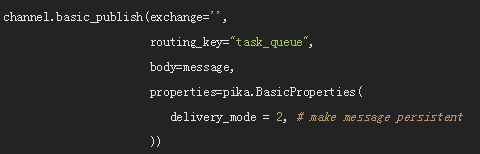


Although this command is correct by itself, it won't work in our setup. That's because we've already defined a queue called hello which is not durable. RabbitMQ doesn't allow you to redefine an existing queue with different parameters and will return an error to any program that tries to do that. But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



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.



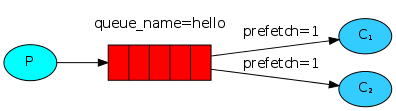
* Note on message persistence

Marking messages as persistent doesn't fully guarantee that a message won't be lost. Although it tells RabbitMQ to save the message to 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 then you can use publisher confirms.

**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.



* 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 use message TTL.

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.

**Publish/Subscribe**

**Prerequisites**

As with other Python tutorials, we will use the Pika RabbitMQ client version 0.11.0.

**What This Tutorial Focuses On**

In the previous tutorial we created a work queue. The assumption behind a work queue is that each task is delivered to exactly one worker. In this part we'll do something completely different -- we'll deliver a message to multiple consumers. This pattern is known as "publish/subscribe".

To illustrate the pattern, we're going to build a simple logging system. It will consist of two programs -- the first will emit log messages and the second will receive and print them.

In our logging system every running copy of the receiver program will get the messages. That way we'll be able to run one receiver and direct the logs to disk; and at the same time we'll be able to run another receiver and see the logs on the screen.

Essentially, published log messages are going to be broadcast to all the receivers.

**Exchanges**

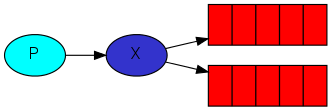
In previous parts of the tutorial we sent and received messages to and from a queue. Now it's time to introduce the full messaging model in Rabbit.

Let's quickly go over what we covered in the previous tutorials:

* A producer is a user application that sends messages.
* A queue is a buffer that stores messages.
* A consumer is a user application that receives messages.

The core idea in the messaging model in RabbitMQ is that the producer never sends any messages directly to a queue. Actually, quite often the producer doesn't even know if a message will be delivered to any queue at all.

Instead, the producer can only send messages to an exchange. An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues. The exchange must know exactly what to do with a message it receives. Should it be appended to a particular queue? Should it be appended to many queues? Or should it get discarded. The rules for that are defined by the exchange type.



There are a few exchange types available: direct, topic, headers and fanout. We'll focus on the last one -- the fanout. Let's create an exchange of that type, and call it logs:



The fanout exchange is very simple. As you can probably guess from the name, it just broadcasts all the messages it receives to all the queues it knows. And that's exactly what we need for our logger.

* **Listing exchanges**

To list the exchanges on the server you can run the ever useful rabbitmqctl:

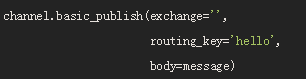


In this list there will be some amq.\* exchanges and the default (unnamed) exchange. These are created by default, but it is unlikely you'll need to use them at the moment.

* **The default exchange**

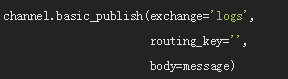
In previous parts of the tutorial we knew nothing about exchanges, but still were able to send messages to queues. That was possible because we were using a default exchange, which we identify by the empty string ("").

Recall how we published a message before:



The exchange parameter is the name of the exchange. The empty string denotes the default or nameless exchange: messages are routed to the queue with the name specified by routing\_key, if it exists.

Now, we can publish to our named exchange instead:



**Temporary queues**

As you may remember previously we were using queues which had a specified name (remember hello and task\_queue?). Being able to name a queue was crucial for us -- we needed to point the workers to the same queue. Giving a queue a name is important when you want to share the queue between producers and consumers.

But that's not the case for our logger. We want to hear about all log messages, not just a subset of them. We're also interested only in currently flowing messages not in the old ones. To solve that we need two things.

Firstly, whenever we connect to Rabbit we need a fresh, empty queue. To do it we could create a queue with a random name, or, even better - let the server choose a random queue name for us. We can do this by not supplying the queue parameter to queue\_declare:



At this point result.method.queue contains a random queue name. For example it may look like amq.gen-JzTY20BRgKO-HjmUJj0wLg.

Secondly, once we disconnect the consumer the queue should be deleted. There's an exclusive flag for that:



**Bindings**



We've already created a fanout exchange and a queue. Now we need to tell the exchange to send messages to our queue. That relationship between exchange and a queue is called a binding.



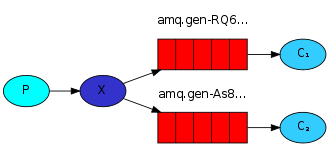
From now on the logs exchange will append messages to our queue.

Listing bindings

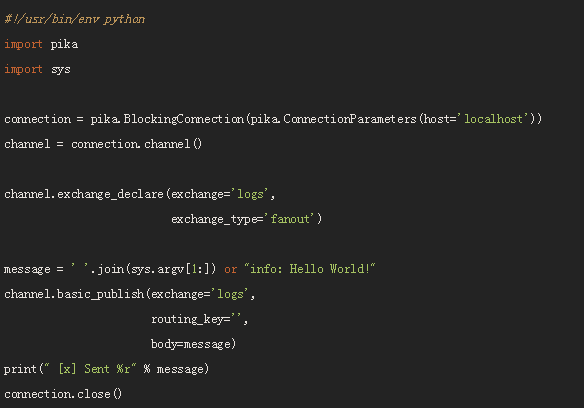
You can list existing bindings using, you guessed it,



**Putting it all together**



The producer program, which emits log messages, doesn't look much different from the previous tutorial. The most important change is that we now want to publish messages to our logs exchange instead of the nameless one. We need to supply a routing\_key when sending, but its value is ignored for fanout exchanges. Here goes the code for emit\_log.py script:



As you see, after establishing the connection we declared the exchange. This step is necessary as publishing to a non-existing exchange is forbidden.

The messages will be lost if no queue is bound to the exchange yet, but that's okay for us; if no consumer is listening yet we can safely discard the message.

The code for receive\_logs.py:



We're done. If you want to save logs to a file, just open a console and type:



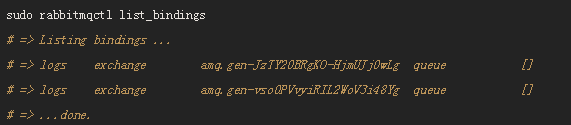
If you wish to see the logs on your screen, spawn a new terminal and run:



And of course, to emit logs type:



Using rabbitmqctl list\_bindings you can verify that the code actually creates bindings and queues as we want. With two receive\_logs.py programs running you should see something like:



The interpretation of the result is straightforward: data from exchange logs goes to two queues with server-assigned names. And that's exactly what we intended.

To find out how to listen for a subset of messages, let's move on to tutorial 4

**Routing**

**Prerequisites**

As with other Python tutorials, we will use the Pika RabbitMQ client version 0.11.0.

**What This Tutorial Focuses On**

In the previous tutorial we built a simple logging system. We were able to broadcast log messages to many receivers.

In this tutorial we're going to add a feature to it - we're going to make it possible to subscribe only to a subset of the messages. For example, we will be able to direct only critical error messages to the log file (to save disk space), while still being able to print all of the log messages on the console.

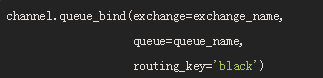
**Bindings**

In previous examples we were already creating bindings. You may recall code like:



A binding is a relationship between an exchange and a queue. This can be simply read as: the queue is interested in messages from this exchange.

Bindings can take an extra routing\_key parameter. To avoid the confusion with a basic\_publish parameter we're going to call it a binding key. This is how we could create a binding with a key:



The meaning of a binding key depends on the exchange type. The fanout exchanges, which we used previously, simply ignored its value.

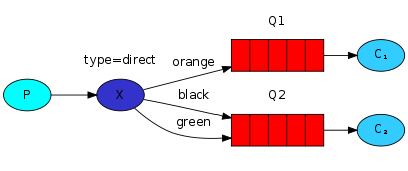
**Direct exchange**

Our logging system from the previous tutorial broadcasts all messages to all consumers. We want to extend that to allow filtering messages based on their severity. For example we may want the script which is writing log messages to the disk to only receive critical errors, and not waste disk space on warning or info log messages.

We were using a fanout exchange, which doesn't give us too much flexibility - it's only capable of mindless broadcasting.

We will use a direct exchange instead. The routing algorithm behind a direct exchange is simple - a message goes to the queues whose binding key exactly matches the routing key of the message.

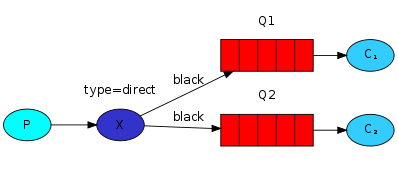
To illustrate that, consider the following setup:



In this setup, we can see the direct exchange X with two queues bound to it. The first queue is bound with binding key orange, and the second has two bindings, one with binding key black and the other one with green.

In such a setup a message published to the exchange with a routing key orange will be routed to queue Q1. Messages with a routing key of black or green will go to Q2. All other messages will be discarded.

**Multiple bindings**



It is perfectly legal to bind multiple queues with the same binding key. In our example we could add a binding between X and Q1 with binding key black. In that case, the direct exchange will behave like fanout and will broadcast the message to all the matching queues. A message with routing key black will be delivered to both Q1 and Q2.

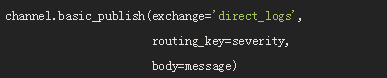
**Emitting logs**

We'll use this model for our logging system. Instead of fanout we'll send messages to a direct exchange. We will supply the log severity as a routing key. That way the receiving script will be able to select the severity it wants to receive. Let's focus on emitting logs first.

Like always we need to create an exchange first:



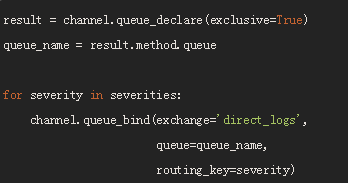
And we're ready to send a message:



To simplify things we will assume that 'severity' can be one of 'info', 'warning', 'error'.

**Subscribing**

Receiving messages will work just like in the previous tutorial, with one exception - we're going to create a new binding for each severity we're interested in.



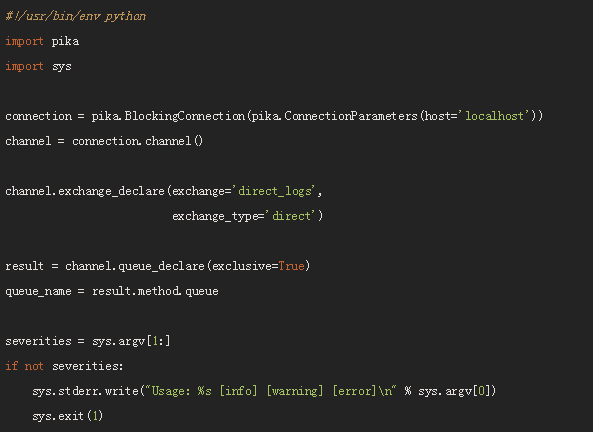
**Putting it all together**

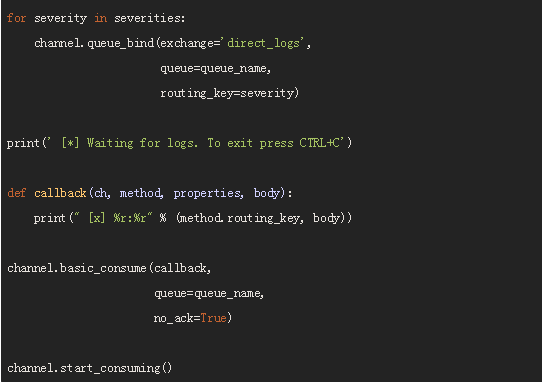


The code for emit\_log\_direct.py:



The code for receive\_logs\_direct.py:





If you want to save only 'warning' and 'error' (and not 'info') log messages to a file, just open a console and type:



If you'd like to see all the log messages on your screen, open a new terminal and do:



And, for example, to emit an error log message just type:



Move on to tutorial 5 to find out how to listen for messages based on a pattern.

**Topics**

**Prerequisites**

As with other Python tutorials, we will use the Pika RabbitMQ client version 0.11.0.

**What This Tutorial Focuses On**

In the previous tutorial we improved our logging system. Instead of using a fanout exchange only capable of dummy broadcasting, we used a direct one, and gained a possibility of selectively receiving the logs.

Although using the direct exchange improved our system, it still has limitations - it can't do routing based on multiple criteria.

In our logging system we might want to subscribe to not only logs based on severity, but also based on the source which emitted the log. You might know this concept from the syslog unix tool, which routes logs based on both severity (info/warn/crit...) and facility (auth/cron/kern...).

That would give us a lot of flexibility - we may want to listen to just critical errors coming from 'cron' but also all logs from 'kern'.

To implement that in our logging system we need to learn about a more complex topic exchange.

**Topic exchange**

Messages sent to a topic exchange can't have an arbitrary routing\_key - it must be a list of words, delimited by dots. The words can be anything, but usually they specify some features connected to the message. A few valid routing key examples: "stock.usd.nyse", "nyse.vmw", "quick.orange.rabbit". There can be as many words in the routing key as you like, up to the limit of 255 bytes.

The binding key must also be in the same form. The logic behind the topic exchange is similar to a direct one - a message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key. However there are two important special cases for binding keys:

* \* (star) can substitute for exactly one word.
* # (hash) can substitute for zero or more words.

It's easiest to explain this in an example:



In this example, we're going to send messages which all describe animals. The messages will be sent with a routing key that consists of three words (two dots). The first word in the routing key will describe a celerity, second a colour and third a species: "<celerity>.<colour>.<species>".

We created three bindings: Q1 is bound with binding key "\*.orange.\*" and Q2 with "\*.\*.rabbit" and "lazy.#".

These bindings can be summarised as:

* Q1 is interested in all the orange animals.
* Q2 wants to hear everything about rabbits, and everything about lazy animals.

A message with a routing key set to "quick.orange.rabbit" will be delivered to both queues. Message "lazy.orange.elephant" also will go to both of them. On the other hand "quick.orange.fox" will only go to the first queue, and "lazy.brown.fox" only to the second. "lazy.pink.rabbit" will be delivered to the second queue only once, even though it matches two bindings. "quick.brown.fox" doesn't match any binding so it will be discarded.

What happens if we break our contract and send a message with one or four words, like "orange" or "quick.orange.male.rabbit"? Well, these messages won't match any bindings and will be lost.

On the other hand "lazy.orange.male.rabbit", even though it has four words, will match the last binding and will be delivered to the second queue.

* **Topic exchange**

Topic exchange is powerful and can behave like other exchanges.

When a queue is bound with "#" (hash) binding key - it will receive all the messages, regardless of the routing key - like in fanout exchange.

When special characters "\*" (star) and "#" (hash) aren't used in bindings, the topic exchange will behave just like a direct one.

**Putting it all together**

We're going to use a topic exchange in our logging system. We'll start off with a working assumption that the routing keys of logs will have two words: "<facility>.<severity>".

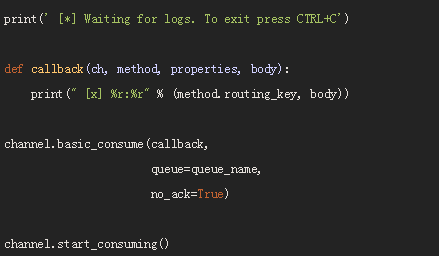
The code is almost the same as in the previous tutorial.

The code for emit\_log\_topic.py:



The code for receive\_logs\_topic.py:





To receive all the logs run:



To receive all logs from the facility "kern":



Or if you want to hear only about "critical" logs:



You can create multiple bindings:



And to emit a log with a routing key "kern.critical" type:



Have fun playing with these programs. Note that the code doesn't make any assumption about the routing or binding keys, you may want to play with more than two routing key parameters.

Move on to tutorial 6 to learn about RPC.

**Remote procedure call (RPC)**

**Prerequisites**

As with other Python tutorials, we will use the Pika RabbitMQ client version 0.11.0.

**What This Tutorial Focuses On**

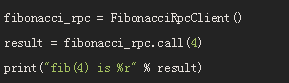
In the second tutorial we learned how to use Work Queues to distribute time-consuming tasks among multiple workers.

But what if we need to run a function on a remote computer and wait for the result? Well, that's a different story. This pattern is commonly known as Remote Procedure Call or RPC.

In this tutorial we're going to use RabbitMQ to build an RPC system: a client and a scalable RPC server. As we don't have any time-consuming tasks that are worth distributing, we're going to create a dummy RPC service that returns Fibonacci numbers.

**Client interface**

To illustrate how an RPC service could be used we're going to create a simple client class. It's going to expose a method named call which sends an RPC request and blocks until the answer is received:



* **A note on RPC**

Although RPC is a pretty common pattern in computing, it's often criticised. The problems arise when a programmer is not aware whether a function call is local or if it's a slow RPC. Confusions like that result in an unpredictable system and adds unnecessary complexity to debugging. Instead of simplifying software, misused RPC can result in unmaintainable spaghetti code.

Bearing that in mind, consider the following advice:

Make sure it's obvious which function call is local and which is remote.

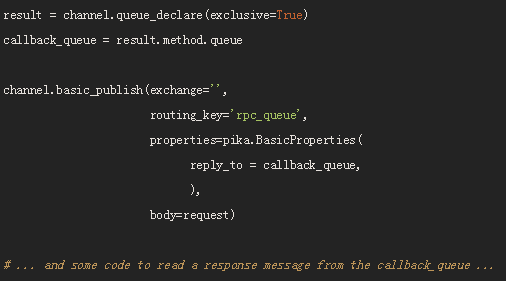
Document your system. Make the dependencies between components clear.

Handle error cases. How should the client react when the RPC server is down for a long time?

When in doubt avoid RPC. If you can, you should use an asynchronous pipeline - instead of RPC-like blocking, results are asynchronously pushed to a next computation stage.

**Callback queue**

In general doing RPC over RabbitMQ is easy. A client sends a request message and a server replies with a response message. In order to receive a response the client needs to send a 'callback' queue address with the request. Let's try it:



* **Message properties**

The AMQP 0-9-1 protocol predefines a set of 14 properties that go with a message. Most of the properties are rarely used, with the exception of the following:

* delivery\_mode: Marks a message as persistent (with a value of 2) or transient (any other value). You may remember this property from the second tutorial.
* content\_type: Used to describe the mime-type of the encoding. For example for the often used JSON encoding it is a good practice to set this property to: application/json.
* reply\_to: Commonly used to name a callback queue.
* correlation\_id: Useful to correlate RPC responses with requests.

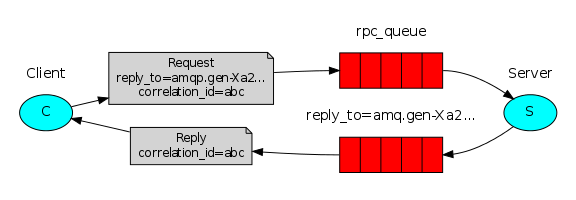
**Correlation id**

In the method presented above we suggest creating a callback queue for every RPC request. That's pretty inefficient, but fortunately there is a better way - let's create a single callback queue per client.

That raises a new issue, having received a response in that queue it's not clear to which request the response belongs. That's when the correlation\_id property is used. We're going to set it to a unique value for every request. Later, when we receive a message in the callback queue we'll look at this property, and based on that we'll be able to match a response with a request. If we see an unknown correlation\_id value, we may safely discard the message - it doesn't belong to our requests.

You may ask, why should we ignore unknown messages in the callback queue, rather than failing with an error? It's due to a possibility of a race condition on the server side. Although unlikely, it is possible that the RPC server will die just after sending us the answer, but before sending an acknowledgment message for the request. If that happens, the restarted RPC server will process the request again. That's why on the client we must handle the duplicate responses gracefully, and the RPC should ideally be idempotent.

**Summary**



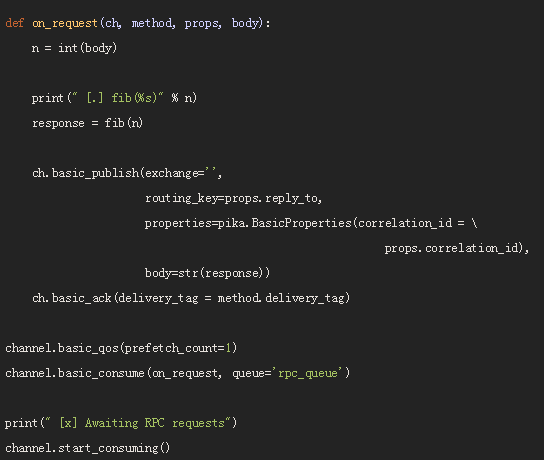
Our RPC will work like this:

* When the Client starts up, it creates an anonymous exclusive callback queue.
* For an RPC request, the Client sends a message with two properties: reply\_to, which is set to the callback queue and correlation\_id, which is set to a unique value for every request.
* The request is sent to an rpc\_queue queue.
* The RPC worker (aka: server) is waiting for requests on that queue. When a request appears, it does the job and sends a message with the result back to the Client, using the queue from the reply\_to field.
* The client waits for data on the callback queue. When a message appears, it checks the correlation\_id property. If it matches the value from the request it returns the response to the application.

Putting it all together

The code for rpc\_server.py:





The server code is rather straightforward:

(4) As usual we start by establishing the connection and declaring the queue.

(11) We declare our fibonacci function. It assumes only valid positive integer input. (Don't expect this one to work for big numbers, it's probably the slowest recursive implementation possible).

(19) We declare a callback for basic\_consume, the core of the RPC server. It's executed when the request is received. It does the work and sends the response back.

(32) We might want to run more than one server process. In order to spread the load equally over multiple servers we need to set the prefetch\_count setting.

The code for rpc\_client.py:





The client code is slightly more involved:

(7) We establish a connection, channel and declare an exclusive 'callback' queue for replies.

(16) We subscribe to the 'callback' queue, so that we can receive RPC responses.

(18) The 'on\_response' callback executed on every response is doing a very simple job, for every response message it checks if the correlation\_id is the one we're looking for. If so, it saves the response in self.response and breaks the consuming loop.

(23) Next, we define our main call method - it does the actual RPC request.

(24) In this method, first we generate a unique correlation\_id number and save it - the 'on\_response' callback function will use this value to catch the appropriate response.

(25) Next, we publish the request message, with two properties: reply\_to and correlation\_id.

(32) At this point we can sit back and wait until the proper response arrives.

(33) And finally we return the response back to the user.

Our RPC service is now ready. We can start the server:



To request a fibonacci number run the client:



The presented design is not the only possible implementation of a RPC service, but it has some important advantages:

* If the RPC server is too slow, you can scale up by just running another one. Try running a second rpc\_server.py in a new console.
* On the client side, the RPC requires sending and receiving only one message. No synchronous calls like queue\_declare are required. As a result the RPC client needs only one network round trip for a single RPC request.

Our code is still pretty simplistic and doesn't try to solve more complex (but important) problems, like:

* How should the client react if there are no servers running?
* Should a client have some kind of timeout for the RPC?
* If the server malfunctions and raises an exception, should it be forwarded to the client?
* Protecting against invalid incoming messages (eg checking bounds) before processing.