4 Solving problems with Rabbit: coding and patterns

This chapter covers

- Designing applications toward messaging
- Messaging patterns
- Fire-and-forget models
- RPC with RabbitMQ

At this point you know how to install, configure, and even run Rabbit in production. It's about time we got to some coding. First, you need to understand problems you're trying to solve when you code messaging into your apps. Like a lot of people who discover RabbitMQ, your lovable authors weren't looking for a message queue; we were looking to solve a decoupling problem. How do you take a time-intensive task and move it out of the app that triggers it (thereby freeing that app to service other requests)? Also, how do you glue together applications written in different languages like PHP and Erlang so that they act as a single system? These seem like

two different problems but they have a common kernel: decoupling the request from the action. Or put another way, both problems demand moving from a synchronous programming model to an asynchronous one.

Normally, when programmers hear *asynchronous programming* they either go running for the hills or think "Cool. Like Node.js right?" Sometimes both. The problem with normal approaches to asynchronous programming is that they're all-or-nothing propositions. You rewrite all of your code so none of it blocks or you're just wasting your time. RabbitMQ gives you a different path. It allows you to fire off a request for processing elsewhere so that your synchronous application can go back to what it was doing. When you start using messaging, you can achieve most of the benefits of pure asynchronous programming without throwing your existing code or knowledge away. In this chapter, we'll show you what asynchronous coding means in the Rabbit world. In particular, we'll show you how to use Rabbit to solve a number of real-world problems from picture processing (parallel processing) and alerting (notifications) to using RabbitMQ for distributed remote procedure calls (APIs) that are as simple as pie. We'll start by teaching some fundamental messaging paradigms and then diving into the code. Let's get decoupling!

4.1 A decoupling story: what pushes us to messaging

You do it all the time. You write your latest and greatest web app (scheduling Chihuahua walking), and decide the fastest way to go is to take web orders and stuff them directly into a database. It makes sense. How much time can stuffing a small record into a database take? Not to mention, it's so simple to code. The problem is, what happens when you go nationwide and you're now scheduling 100,000 Chihuahuas an hour? Or better yet, you decide you want to store your data in two places (gotta archive those requests). Guess what? It's time to rip out all of that carefully debugged code. Coupling an app directly to storage is usually a recipe for rip-and-replace later, and that's where messaging can help you.

4.1.1 An asynchronous state of mind (separating requests and actions)

How often do you operate in a synchronous fashion? If you order a pizza, do you wait for it to show up before you do anything else? Of course not. You watch TV or read a book, or maybe give your sweetie some quality conversation time. Rarely do you put your life on hold waiting for a response to your requests. You multitask, so your lives can scale and you can get more done. Your apps need the same approach.

Why do you design your apps to be synchronous in the first place? Mostly, because you think about the whole job instead of the smaller tasks that make it up. You think "my app needs to schedule a Chihuahua appointment." Instead the reality is that your app needs to receive a scheduling request; then it needs to store that request in a database; then it has to alert the closest dog walker; and finally it needs to let the customer know they're scheduled. Even if you make your app multithreaded, you've severely limited the rate at which you can take orders because each thread has to wait for the record to be stored and the dog walker to be alerted. Rather, you should look at those four steps

as falling into two separate apps: an app that takes the request, and an app that processes the request. To hijack a great analogy from Gregor Hohpe, we could call it the Coffee Bean model (Coffee Bean & Tea Leaf is a chain of coffee houses in California).

When you place an order for your chai latte, you don't wait at the cash register until your order is ready. Instead, Coffee Bean splits the order taking operation from the order preparation operation. The order taker collects your request (and your dinero), and transmits a message to the baristas telling them what you ordered. You then wait for your order to be prepared, freeing up the order taker to take another order. The most important part of the operation is getting your money collected, and so by separating order taking from order processing, Coffee Bean has maximized the number of orders they can take per minute. Similarly, if the backlog of coffee waiting to be prepared gets too high, they can add more baristas to reduce the backlog without changing the number of order takers. By decoupling the process (separating requests and actions), they've increased the amount of work they can accomplish with the same number of workers and made it easy to scale up when they need to. Messaging does the same thing for your app.

So let's reanalyze your Chihuahua app with decoupling in mind. Figure 4.1 shows the steps in completing a dog walking order.

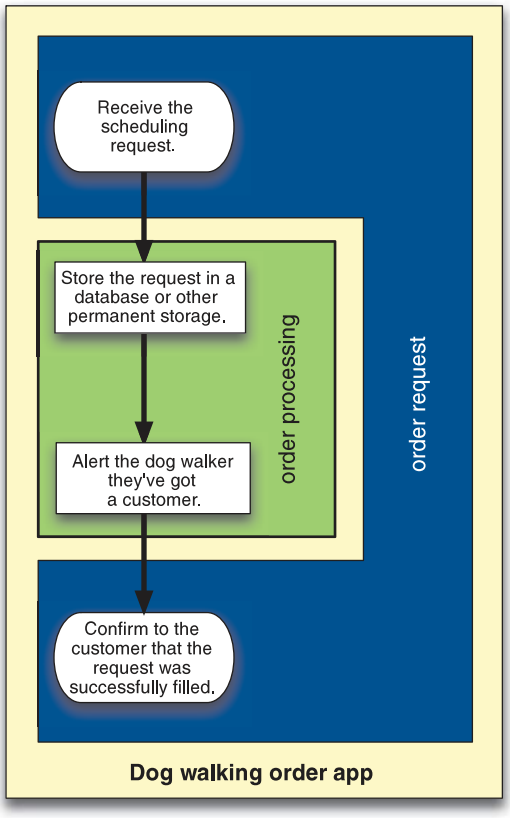


Figure 4.1 Steps for completing a dog walking order

If you want to increase the scalability and flexibility of your app, you need to split it into two different apps: *dog _walk_order* and *dog_walk_process*, as in figure 4.2. dog_walk_order sits on the internet and receives web requests to schedule walkings. When it receives a request, dog_walk_order creates a new AMQP message and publishes that into the `chihuahua_scheduling` exchange on Rabbit. dog_walk_order can then put Customer A on hold and go receive other requests. Meanwhile, dog_walk_process listens to a Rabbit queue and receives the message containing customer A's scheduling order. It then gets to work creating the required database entry for the order and firing off a text message to your main dog walker, Gustav. Once Gustav has been sent his text message, dog _walk_process sends a message back

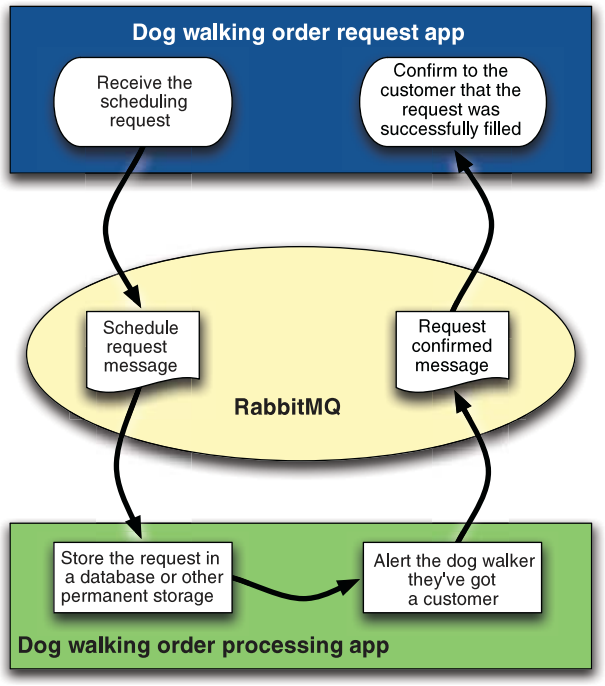


Figure 4.2 Splitting the dog walking program into two apps

to dog_walk_order that customer A's request was successfully processed. During the whole time that dog_walk_process was dealing with Customer A's order, dog_walk _order was able to receive 100 new walking requests. Had they still been one app, you would've received only one walking request during that whole interval.

By putting RabbitMQ between two parts of your app that were once tightly coupled, you've made it possible to continually receive requests, where before you could only process one at a time. But you've also opened up a whole other world of benefits. What if you get so much load that one order processing server is no longer enough?

4.1.2 Affording scale: a world without load balancers

One of the great things about using messaging is that it's simple to add processing capacity to your apps. Let's say you've just expanded your service to Japan, and now you're receiving 1,000,000 walking orders a second. Though your frontend order taker is more than able to keep up with the load, your order processor is keeling over. Taking a customer's order no longer keeps you from taking other orders, but those customers are getting fed up waiting for you to confirm their reservations. You need more order processors. Just like at Coffee Bean, you can add more baristas. In your case, you spool up additional dog_walk_process servers and attach them to the queue that receives the orders. Presto! Without one line of code change, you added 10x processing capacity by spooling up 10 new dog_walk_process servers. The best part is that RabbitMQ will evenly distribute the requests among the processing servers due to the automatic round-robin behavior we talked about in section 2.2. No expensive load balancers required.

That's important for any organization, not just cash-conscious startups. Load balancing hardware is expensive, which means you're normally limited to how many places you can use it to decouple and scale your apps. If instead you can use AMQP and Rabbit, then you can add decoupling and load balancing anywhere you want for free. Not to mention, you can do more complex routing, such as send a message to more than one destination in addition to round-robin load balancing. Load balancers will always have a place on the frontend distributing requests coming in from the internet, but if you can heavily leverage messaging, you can reduce your reliance on them inside the firewall and greatly increase the number of places you decouple your apps. Decoupled apps are scalable apps.

4.1.3 Zero-effort APIs: why be locked into just one language?

We've skipped over one of the best benefits of using AMQP to decouple your apps: APIs for free. Today everyone is talking about web APIs that allow you to integrate an app's functionality into any other app. Generally, this takes a bit of effort because you end up writing a lot of code to translate incoming HTTP requests into your app's function calls. If you write your app using AMQP to connect the parts, you actually get an API for no additional effort—an API that uses messaging.

Let's say you've expanded your Chihuahua walking business into dog washing. You have two new apps to support the new service: *dog_wash_request* and *dog_wash_process*. Then you get a great idea: offer a free dog walk with every wash. Since both the washing and walking scheduling apps use AMQP, all you need to do is update dog_wash _request to generate an additional AMQP message that contains the dog walk scheduling information. dog_wash_request can instantly take advantage of dog_walk_process. This means no recoding of the scheduling code and no need to duplicate that code inside the dog washing apps. Equally important, there's no requirement that the walking and washing apps be written in the same language.

When you wrote the dog walking apps, you may have chosen Erlang as the best language for the job. But in the months since, you've discovered how much you like Clojure for building high-concurrency applications. So you wrote the dog washing apps in Clojure. If you were using Erlang's built-in communication protocol for connecting dog_walk_request to dog_walk_process, it'd be difficult for the dog washing apps to talk to dog_walk_process since they're not written in Erlang. But because AMQP is language-agnostic and has native language bindings for dozens of languages, you can easily connect a Clojure request receiver to an Erlang request processor over Rabbit. Using AMQP to connect your applications gives you the flexibility to use the right language for each part of the job, and even to change your mind later and connect in new applications written in completely different languages. RabbitMQ makes it easy to connect any and all parts of your infrastructure in any way you want.

So, the first thing you should always ask is, how can break you apps apart? Or rather, which parts of your app are order takers and which parts are order processors? With that in mind, let's dive into some real-world examples of using Rabbit and messaging to solve real problems and answer those questions.

4.2 Fire-and-forget models

When we look at the types of problems messaging can solve, one of the main areas that messaging fits is fire-and-forget processing. Whether you need to add contacts to a mailing list or convert 1,000 pictures into thumbnails, you're interested in the jobs getting done but there's no reason they need to be done in real-time. In fact, you usually want to avoid blocking the user who triggered the jobs. We describe these types of jobs as *fire-and-forget*: you create the job, put it on the exchange, and let your app get back to what it was doing. Depending on your requirements, you may not even need to notify the user when the jobs complete.

Two general types of jobs fit into this pattern:

- *Batch processing*—Work and transformations that need be completed on a large data set. This can be structured as a single job request or many jobs operating on individual parts of the data set.
- *Notifications*—A description of an event that has occurred. This can be anything from a message to be logged, to an actual alert that should be sent to another program or an administrator.

We're going to show you two different real-world examples of fire-and-forget apps that fit into these two categories. The first is an alerting framework that will allow the apps in your infrastructure to generate administrator alerts without worrying about where they need to go or how to get them there. The second example is a perfect demonstration of batch processing: taking a single image upload and converting into multiple image sizes and formats. When you're done with this section you'll have the most fundamental type of RabbitMQ programming under your belt: triggering work with messages that need no reply. Let's start generating some alerts!

4.2.1 Sending alerts

No matter what type of apps you write, getting notifications when things go awry is critical. Typically you run some sort of service monitor like Nagios to let you know when your app is down or services that it relies upon are unavailable. But what about getting notified when your app is experiencing an unusual number of requests for user logins, all from a single IP? Or perhaps you'd like to allow your customers to be notified when unusual events occur to their data? What you need is for your app to generate alerts, but this opens up a whole new set of questions and adds a lot of complexity to your app. What events do you alert on, and more important, how do you alert? SMS? IM? No matter how you slice it, you're looking at adding a lot of new surface area to your code for bugs to hide in. For example, what happens when the SMS gateway is down? All of your web apps that need to alert now need error-handling code to deal with the SMS server being unavailable.

Worry not, for RabbitMQ is riding to your rescue. The only thing about alerting that inherently needs to be done in your web apps is generating the contents of the alert. There's no reason why your web app needs to know whom the alert should go

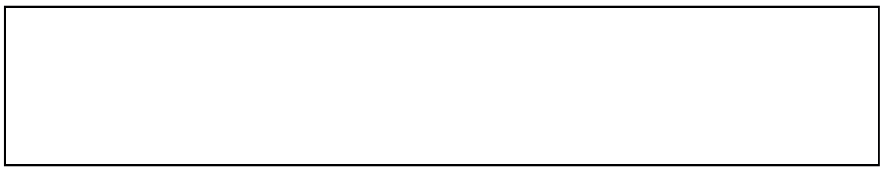
to, how to get it there, or what to do when the alert deliveries go awry. All you need to do is write a new alerting server app that receives alert messages via Rabbit, and then enhance your web app to publish these alert messages where appropriate.

How should you design this new alerting framework? Particularly, what type of AMQP exchange should you use? You could use a fanout exchange and then create a queue for each alert transmission type (IM, Twitter, SMS, and so on). The advantage is that your web app doesn't have to know anything about how the alerts will be delivered to the ultimate receiver. It just publishes and moves on. The disadvantage is that *every* alert transmitter gets a copy, so you get flooded with an IM, a text message, and a Twitter direct message every time an alert happens.

A better way to organize your alerting system would be to create three severity levels for your alerts: info, warning, and critical. But with the fanout exchange, any alert published would get sent to all three severity level queues. You could instead create your exchange as a direct exchange, which would allow your web app to tag the alert messages with the severity level as the routing key. But what would happen if you chose a topic exchange? Topic exchanges let you create flexible tags for your messages that target them to multiple queues, but only the queues providing the services you want (unlike the fanout exchange). If you were to use a topic exchange for your alerting framework, you wouldn't be limited to just one severity level per alert. In fact, you could now tag your messages not only with a severity level, but also the type of alert it is. For example, let's say Joe Don Hacker is hitting your statistics server with 10,000 requests per second for map data on your dog walking reservations. In your organization, you need an alert about this to go both to the infrastructure admins (who get all alerts flagged as `critical`), and to your API dev team (who get all alerts tagged `rate_limiting`). Since you've chosen a topic exchange for the alerting framework, your web app can tag the alert about such underhanded activity with `critical.rate_limiting`. Presto! The alert message is automatically routed by RabbitMQ to the `critical` and `rate_limiting` queues, because of the exchange bindings you've created: `critical.*` and `*.rate_limiting`. Figure 4.3 shows how the flow of your alerting system will work.

To build this alerting framework you'll need the Pika library you installed as a part of your Hello World in chapter 2. If you skipped that part, here are some quick steps to get Pika installed (assuming you don't have easy_install yet either):

```
$ wget http://peak.telecommunity.com/dist/ez_setup.py
...
  (25.9 KB/s) - ez_setup.py saved [10285/10285]

$ python ez_setup.py
...
Installed /Library/Python/2.6/site-packages/setuptools-0.6...
$ easy_install pika
...
Installed /Library/Python/2.6/site-packages/pika-0.9.6-py2.6.egg
Processing dependencies for pika
Finished processing dependencies for pika
```



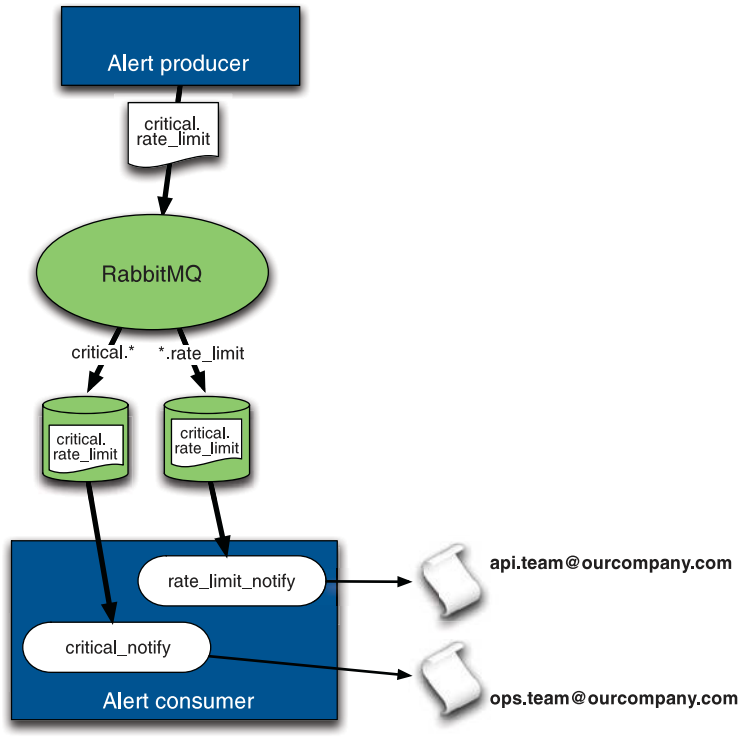


Figure 4.3
Alerting system flow

Next you need to set up the RabbitMQ user and password the applications will use to publish and receive alert messages. Let's call the user `alert_user` and give it the password `alertme`. Let's also grant `alert_user` read/write/configure permissions on the default vhost `/`.

From the `./sbin` directory of your RabbitMQ install, run the following:

```
$ ./rabbitmqctl add_user alert_user alertme
Creating user "alert_user" ...
...done.
$ ./rabbitmqctl set_permissions alert_user ".*" ".*" ".*"
Setting permissions for user "alert_user" in vhost "/" ...
...done.
```

With the setup out of the way, you're ready to work on the most important part of your alerting system: the AMQP consumer that will receive the alert messages and transmit them to their destinations. Create a file called `alert_consumer.py` and put the code in the following listing inside.

Listing 4.1 Connect to the broker

```
import json, smtplib
import pika

if __name__ == "__main__":
    AMQP_SERVER = "localhost"
```

Broker settings

```
AMQP_USER = "alert_user"
AMQP_PASS = "alertme"
AMQP_VHOST = "/"
AMQP_EXCHANGE = "alerts"

creds_broker = pika.PlainCredentials(AMQP_USER, AMQP_PASS)
conn_params = pika.ConnectionParameters(AMQP_SERVER,
                                        virtual_host = AMQP_VHOST,
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)

channel = conn_broker.channel()
```

Establish connection to broker

The first thing this code does is import the libraries you'll need to make the consumer tick, and tell Python where the main body of your program is (`if __name__ == "__main__":`). Next, you establish the settings you need to make a successful connection to your broker (user, name, password, virtual host, and so forth). The settings assume you have RabbitMQ running locally on your development workstation and are using the username and password you just created. For simplicity, let's use the default virtual host `/` where you're going to create an exchange called `alerts`. Here's where the real action starts:

```
channel.exchange_declare( exchange=AMQP_EXCHANGE,
                          type="topic",
                          auto_delete=False)
```

You're declaring the `alerts` exchange as a topic exchange with the `type="topic"` parameter that's passed to `channel.exchange_declare`. The `auto_delete` parameter you're also passing to the exchange and queue declarations ensures they'll stick around when the last consumer disconnects from them.

Remember that we talked about two tagging patterns for alerts:

- `.*` for tagging alerts with their severity level (say, `critical`)
- `*.` for tagging alerts with a specific alert type such as `rate_limiting`

What you need to do is create bindings that implement these rules so that the alert messages go to the queues you want. For your example, let's create a binding that routes any messages with tags starting with `critical.` to the `critical` queue. Let's also create a different binding that routes any messages with tags ending in `.rate_limit` to the `rate_limit` queue. Go ahead and create the `critical` and `rate_limit` queues and bind them, as shown in the following listing.

Listing 4.2 Declare and bind queues and exchanges for the alert topics

```
channel.queue_declare(queue="critical", auto_delete=False)
channel.queue_bind(queue="critical",
                   exchange="alerts",
                   routing_key="critical.*")

channel.queue_declare(queue="rate_limit", auto_delete=False)
channel.queue_bind(queue="rate_limit",
                   exchange="alerts",
                   routing_key="*.rate_limit")
```

critical queue and critical.* topic binding

rate_limit queue and *.rate_limit topic binding

You'll notice that the binding rule created for critical alerts is `critical.*` and not `critical*`. This is because RabbitMQ uses `.` as a separator for different parts of a tag. If you were to use `critical*` as the rule, only messages tagged exactly `critical*` would match. What you want is to match `critical.mywebapp`, `critical.rate_limit`, or anything that starts with `critical.`; hence the binding rule should be `critical.*`. When using topic exchanges, it's important to be careful to design your tagging patterns to use `.` to separate the parts of the tag you want to match separately on.

You could've also passed `durable=True` to the queue declarations and bindings, which would ensure that they survived a restart of RabbitMQ. Since restarting your consumer will automatically create the exchange, queues, and bindings it needs, you don't need to worry about durability for your alerting system. The other reason you're not concerned about making the queues durable is because you're not going to flag your alert messages as durable either. Your system could be handling very high volumes of alerts, so you want to ensure the highest performance and not use durable messaging, which is persisted to relatively slow disk.

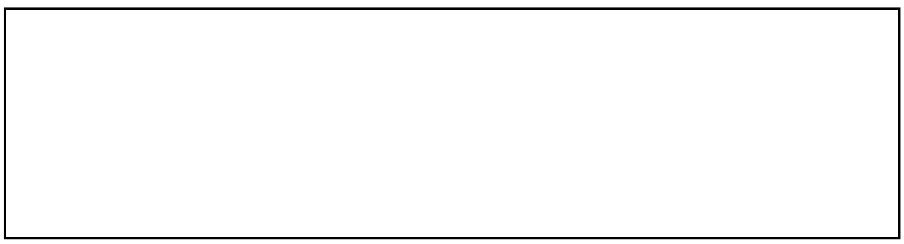
You might be thinking, "We have exchanges, queues, and bindings … where do we turn an alert message into an actual alert?" You do that by setting up your consumer subscriptions and starting the listener loop, as in the following listing.

Listing 4.3 Attach the alert processors

```
channel.basic_consume( critical_notify,
                       queue="critical",
                       no_ack=False,
                       consumer_tag="critical")

channel.basic_consume( rate_limit_notify,
                       queue="rate_limit",
                       no_ack=False,
                       consumer_tag="rate_limit")

print "Ready for alerts!"
channel.start_consuming()
```

Let's take the `channel.basic_consume` call apart and explain what each parameter does:

- `critical_notify` is the callback. It's the function that will be called when a message is received for your subscription to the `critical` queue. The Pika library will call `critical_notify` when a message is received on this subscription, passing in the channel, message headers, message body, and message method from the message.
- `queue="critical"` specifies the queue you want to receive messages from.
- `no_ack=False` tells RabbitMQ you want to explicitly acknowledge received messages. This will keep Rabbit from sending new messages from the queue until you've processed and acknowledged the last one you received.
- `consumer_tag` is an identifier that will identify this subscription uniquely on the AMQP channel you created with `channel = conn_broker.channel()`. The



consumer tag is what you'd pass to RabbitMQ if you wanted to cancel your subscription.

Once you've established the consumer subscriptions, you only need to call `channel.start_consuming()` to start your consumer listening for messages. You may have noticed that the callback functions (`critical_notify` and `rate_limit_notify`) you specified for your subscriptions haven't been defined yet. Let's go ahead and specify one of those in the following listing.

Listing 4.4 Critical alerts processor

```
def critical_notify(channel, method, header, body):
    """Sends CRITICAL alerts to administrators via e-mail."""

    EMAIL_RECIPS = ["ops.team@ourcompany.com",]

    message = json.loads(body)

    send_mail(EMAIL_RECIPS, "CRITICAL ALERT", message)
    print ("Sent alert via e-mail! Alert Text: %s  " + \
           "Recipients: %s") % (str(message), str(EMAIL_RECIPS))

    channel.basic_ack(delivery_tag=method.delivery_tag)
```

Decode message from JSON

Transmit email to SMTP server

Acknowledge message

When a consumer callback is called, Pika passes in four parameters related to the message:

- `channel`—The channel object you're communicating on with Rabbit. If you have multiple channels open, it'll be the one associated with the subscription the message was received on.
- `method`—A method frame object that carries the consumer tag for the related subscription and the delivery tag for the message itself.
- `header`—An object representing the headers of the AMQP message. These carry optional metadata about the message.
- `body`—The actual message contents.

In `critical_notify` the first thing to check is the `content_type` header. Your alerts will be JSON encoded, so you'll check the content type to make sure it's `application/json`. The `content_type` is optional, but it's useful when you want to communicate encoding information about the message between producer and consumer. After you've verified the content type, you decode the message body from JSON to text and construct an email to the Ops Team (ops.team@ourcompany.com) containing the alert text. Once the email alert has been successfully sent, you send an acknowledgement back to RabbitMQ that you've received the message. The acknowledgement is important because RabbitMQ won't give you another message from the queue until you've acknowledged the last one you received. By putting the acknowledgement as the last operation, you ensure that if your consumer were to crash, RabbitMQ would assign the message to another consumer.

With all of the pieces of your consumer explained, let's look at the whole thing put together in the following listing.

Listing 4.5 Alert consumer—alert_consumer.py, start to finish

```
import json, smtplib
import pika

def send_mail(recipients, subject, message):
    """E-mail generator for received alerts."""
    headers = ("From: %s\r\nTo: \r\nDate: \r\n" + \
               "Subject: %s\r\n\r\n") % ("alerts@ourcompany.com",
                                         subject)

    smtp_server = smtplib.SMTP()
    smtp_server.connect("mail.ourcompany.com", 25)
    smtp_server.sendmail("alerts@ourcompany.com",
                         recipients,
                         headers + str(message))
    smtp_server.close()

def critical_notify(channel, method, header, body):
    """Sends CRITICAL alerts to administrators via e-mail."""

    EMAIL_RECIPS = ["ops.team@ourcompany.com",]

    message = json.loads(body)

    send_mail(EMAIL_RECIPS, "CRITICAL ALERT", message)
    print ("Sent alert via e-mail! Alert Text: %s  " + \
           "Recipients: %s") % (str(message), str(EMAIL_RECIPS))

    channel.basic_ack(delivery_tag=method.delivery_tag)

def rate_limit_notify(channel, method, header, body):
    """Sends the message to the administrators via e-mail."""

    EMAIL_RECIPS = ["api.team@ourcompany.com",]

    message = json.loads(body)

    #(f-asc_10) Transmit e-mail to SMTP server
    send_mail(EMAIL_RECIPS, "RATE LIMIT ALERT!", message)

    print ("Sent alert via e-mail! Alert Text: %s  " + \
           "Recipients: %s") % (str(message), str(EMAIL_RECIPS))

    channel.basic_ack(delivery_tag=method.delivery_tag)

if __name__ == "__main__":
    AMQP_SERVER = "localhost"
    AMQP_USER = "alert_user"
    AMQP_PASS = "alertme"
    AMQP_VHOST = "/"
    AMQP_EXCHANGE = "alerts"

    creds_broker = pika.PlainCredentials(AMQP_USER, AMQP_PASS)
    conn_params = pika.ConnectionParameters(AMQP_SERVER,
                                            virtual_host = AMQP_VHOST,
                                            credentials = creds_broker)
    conn_broker = pika.BlockingConnection(conn_params)

    channel = conn_broker.channel()

    channel.exchange_declare( exchange=AMQP_EXCHANGE,
```

Notify processors

Decode message from JSON

Transmit email to SMTP server

Acknowledge message

Decode message from JSON

Broker settings

Acknowledge message

Establish connection to broker

Declare the exchange

Build queues and bindings for topics

```
                              type="topic",
                              auto_delete=False)

    channel.queue_declare(queue="critical", auto_delete=False)
    channel.queue_bind(queue="critical",
                       exchange="alerts",
                       routing_key="critical.*")
    channel.queue_declare(queue="rate_limit", auto_delete=False)
    channel.queue_bind(queue="rate_limit",
                       exchange="alerts",
                       routing_key="*.rate_limit")

    channel.basic_consume( critical_notify,
                           queue="critical",
                           no_ack=False,
                           consumer_tag="critical")

    channel.basic_consume( rate_limit_notify,
                           queue="rate_limit",
                           no_ack=False,
                           consumer_tag="rate_limit")

    print "Ready for alerts!"
    channel.start_consuming()
```

Make alert processors

You now have an elegant consumer that will translate alert AMQP messages into email alerts targeted at different groups simply by manipulating the message tag. Adding additional alert types and transmission methods is simple. All you need to do is create a consumer callback to provide the new alert processing and connect it to a queue that's populated via a binding rule for the new alert type. Your consumer wouldn't be very useful without alerts for it to process. So let's see what it takes to produce alerts that your consumer can act on.

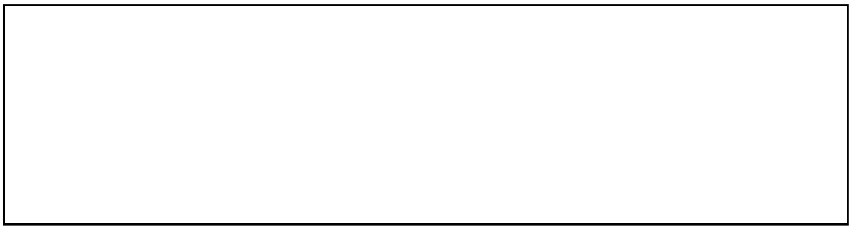
Our goal when we started this section was to make producing alerts simple and uncomplicated for existing apps. If you look at the following listing, you'll see that, though the consumer takes some 90 lines of code to process an alert, the alert itself can be generated in less than 20 lines.

Listing 4.6 Alert generator example—alert_producer.py

```
import json, pika
from optparse import OptionParser

opt_parser = OptionParser()
opt_parser.add_option("-r",
                      "--routing-key",
                      dest="routing_key",
                      help="Routing key for message " + \
                      " (e.g. myalert.im)")
opt_parser.add_option("-m",
                      "--message",
                      dest="message",
                      help="Message text for alert.")

args = opt_parser.parse_args()[0]

creds_broker = pika.PlainCredentials("alert_user", "alertme")
```

Read in command-line arguments

Establish connection to broker



```
conn_params = pika.ConnectionParameters("localhost",
                                        virtual_host = "/",
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)

channel = conn_broker.channel()

msg = json.dumps(args.message)
msg_props = pika.BasicProperties()
msg_props.content_type = "application/json"
msg_props.durable = False

channel.basic_publish(body=msg,
                      exchange="alerts",
                      properties=msg_props,
                      routing_key=args.routing_key)

print ("Sent message %s tagged with routing key '%s' to " + \
       "exchange '/'.") % (json.dumps(args.message),
                           args.routing_key)
```

Publish alert message to broker

The sample producer can be run from the command line to generate alerts with any contents and routing tags you like. The first part of the program simply extracts the message and the routing key from the command line. From there you're connecting to the RabbitMQ broker identically to the way you did in the alert consumer. Where things get interesting is when you publish the message:

```
msg = json.dumps(args.message)
msg_props = pika.BasicProperties()
msg_props.content_type = "application/json"
msg_props.durable = False

channel.basic_publish(body=msg,
                      exchange="alerts",
                      properties=msg_props,
                      routing_key=args.routing_key)
```

Five lines of code is all it takes for you to create the alert message and tag it with the appropriate routing key (say, `critical.mywebapp`). After you JSON-encode the alert's message text, you create a `BasicProperties` object called `msg_props`. This is where you can set the AMQP message's optional content type header, and also where you'd make the message durable if you wanted persistency. Finally, in one line of code you publish the message to the `alerts` exchange with the routing key that classifies what type of alert it is. Since messages with routing keys that don't match any bindings will be discarded, you can even tag alerts with routing keys for alert types you don't support yet. As soon as you do support those alert types, any alert messages with those routing keys will be routed to the right consumer. The last bit to note about the consumer is the `block_on_flow_control` flag you're passing to `channel.basic_publish`. This tells Pika to hold off on returning from `basic_publish` if RabbitMQ's flow control mechanism tells it to stop publishing. When RabbitMQ tells Pika it's okay to proceed, it'll finally return, allowing more publishing to occur. This makes your producer play nicely with RabbitMQ so that if Rabbit becomes overloaded, it can throttle the

producer to slow it down. If you're publishing alerts from another program that can't afford to be blocked, be sure to set `block_on_flow_control` to false.

In only 100 lines of code total, you've given your web apps a flexible and scalable way to issue alerts that then get transmitted asynchronously to their recipients. You've also seen how beneficial the fire-and-forget messaging pattern can be when you need to transmit information to be processed quickly but don't need to know the result of the processing. For example, you could easily extend the alert consumer to add an additional processor that uses the binding pattern `*.*` to log a copy of all alerts to a database. But alerting and logging are far from the only uses of the fire-and-forget messaging pattern. Let's look at an example where you need to perform CPU-intensive processing on the contents of the message, and how RabbitMQ can help you move that into an asynchronous operation.

4.2.2 Parallel processing

Say you started running your own social network website and you just deployed a shiny new feature: picture uploads. People want to share their holiday pictures with friends and family—perhaps you've seen this somewhere. Also, to improve the interaction among users, you want to notify their friends when one of their contacts has uploaded a new picture. A week after the new feature release, the marketing guys come to your desk asking you to give some *points* to the users, a reward for the pictures they upload to encourage them to keep submitting pictures and improve the activity on the site. You agree and add a few lines of code, and now you hook a *reward system* into the upload picture process. It looks a bit nasty for your coder eyes, but it's working as expected and the boss is happy with the results.

Next month the bandwidth bill arrives and the ops guy is angry because the bandwidth usage has doubled. The external API offered to clients is displaying full-size images when it should be offering links to small thumbnails. So you'd better get your uploading code generating those thumbnails too. What to do? The easy way would be to add one more hook in there and execute the thumbnail generation directly from the upload controller, but wait … If for every picture upload you have to execute a picture resize operation, this means the frontend web servers will get overloaded, so you can't just do that. And users of your website don't want to wait for your picture processing script to get a confirmation that their upload is okay. This means you need a smarter solution, something that allows you to run tasks in parallel and in a different machine than the one serving requests to the users.

You can see that resizing a picture, rewarding the user, and notifying friends are three separate tasks. Those tasks are independent in that they don't have to wait for each other's results to be able to run, which means that you can refactor your code not only to process the image resize separately, but also to do those other things in parallel. Also, if you achieve such design, you can cope with new requirements easily. You need to log every picture upload? You just add a new worker to do the logging, and so on.

This sounds nice, almost like a dream, but all this parallelization stuff seems hard to accomplish. How much do you have to code to achieve message multicast? Not much; enter the *fanout* exchange.

As we said when we described the exchange types, the *fanout* exchange will put a copy of the message into the bound queues, as simple as that, and that's what you need for your upload picture module. Every time the user uploads a picture, instead of doing all the processing work right away, you'll publish a message with the picture metainformation and then let your asynchronous workers do the rest in parallel. RabbitMQ will ensure that each consumer gets a copy of the message. It's the worker's duty to process it accordingly.

The messages will contain the following metainformation about the picture: the *image ID* of the picture, the *user ID*, and the *path* to locate the picture on the filesystem. You'll use JSON as the data exchange format. This will make it easier in the future if you need to support several languages for the different tasks. Your messages will look like this:

```
{
    'image_id': 123456,
    'user_id': 6543,
    'image_path': '/path/to/pic.jpg'
}
```

Figure 4.4 shows that you'll declare an `upload-pictures` exchange and will bind three queues to it: `resize-picture`, `add-points`, and `notify-friends`. From this design you can tell that adding a new *kind* of task, like logging, is just a matter of declaring a new queue and binding it to the `upload-pictures` exchange. Your focus as developers will be to code each of the workers and the publishing logic; RabbitMQ will do the rest.

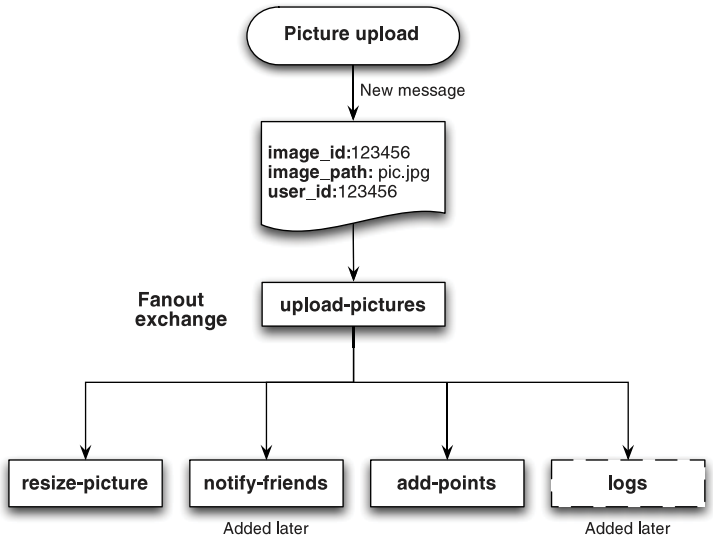
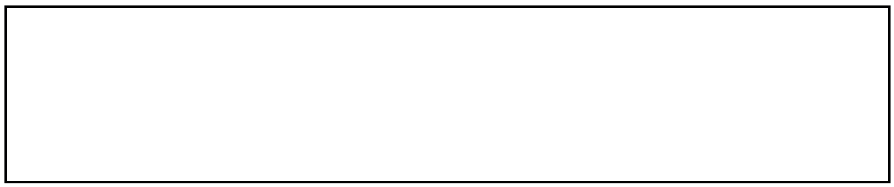


Figure 4.4 Uploading pictures



So, let's start by adding the publisher logic into the upload picture module, as in the following listing. You omit the logic for taking the picture from the POST request and moving it to some place on the filesystem.

Listing 4.7 Upload pictures publisher

```
<?php

$channel->exchange_declare('upload-pictures',
  'fanout', false, true, false);

$metadata = json_encode(array(
      'image_id' => $image_id,
      'user_id' => $user_id,
      'image_path' => $image_path
      ));

$msg = new AMQPMessage($metadata,
                array('content_type' => 'application/json',
                    'delivery_mode' => 2));

$channel->basic_publish($msg, 'upload-pictures');
?>
```

1 Declare exchange

2 Encode image metadata as JSON

3 Instantiate AMQP

4 Publish message

Let's see what you did here. The code for obtaining an AMQP channel isn't present since we covered that in previous examples. At 1 you declare the `upload-pictures` exchange, with a `fanout` type and with *durable* properties. Then at 2 you create the message metadata encoded as JSON. The `$image_id`, `_$user_id`, and `$image_path` were initialized during the upload process. At 3 you create a new instance of the message specifying the `deliver_mode` as `2` to make it persistent. Finally at 4 you publish the message to the `upload-pictures` exchange. You don't need to provide a routing key since the messages will be fanned-out to all the bound queues.

Next let's create one of the consumers, the one for adding points to the users after each upload. Check inside `add-points-consumer.php` for the complete code, since the following listing omits bits that we've covered before, like including the AMQP libraries or instantiating the connection and the channel.

Listing 4.8 Add points consumer

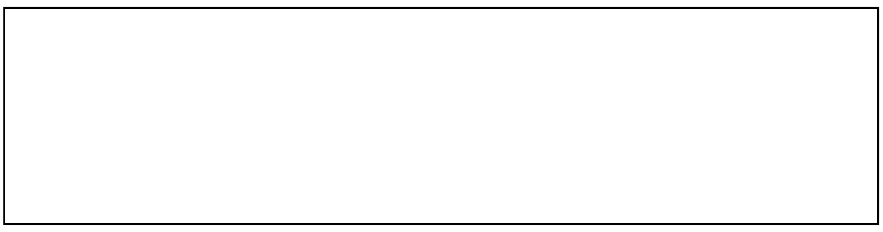
```
<?php

$channel->exchange_declare('upload-pictures',
    'fanout', false, true, false);

$channel->queue_declare('add-points',
    false, true, false, false);

$channel->queue_bind('add-points', 'upload-pictures');

$consumer = function($msg){};

$channel->basic_consume($queue,
                        $consumer_tag,
                        false,
```

1 Declare exchange

2 Declare queue

3 Bind queue

4 Code omitted



```
                    false,
                    false,
                    false,
                    $consumer);
?>
```

5 Start consuming messages

The code is straightforward. At 1 you declare the *topic* exchange as when publishing the message; then at 2 you create the `add-points` queue where the message will be delivered by RabbitMQ. You bind that queue at 3 to the exchange using the empty routing key. At 4 you omit the code for your callback function for now; at 5 you send the `basic_consume` command to prepare the consumer. You also omit the wait loop and the channel and connection cleanup code. The following listing shows the callback function.

Listing 4.9 Add points callback function

```
<?php

function add_points_to_user($user_id){
    echo sprintf("Adding points to user: %s\n", $user_id);
}

$consumer = function($msg){

    if($msg->body == 'quit'){
        $msg->delivery_info['channel']->
            basic_cancel($msg->delivery_info['consumer_tag']);
    }

    $meta = json_decode($msg->body, true);

    add_points_to_user($meta['user_id']);

    $msg->delivery_info['channel']->
        basic_ack($msg->delivery_info['delivery_tag']);
};

?>
```

1 Add points to user function

Consumer callback 2

3 Stop consuming messages

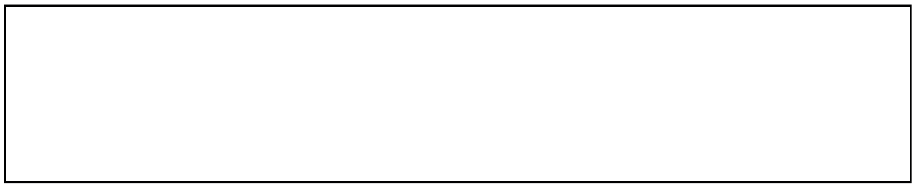
Decode JSON metadata 4

5 Process data

Acknowledge message

In listing 4.9 you have the code for actually processing the message. At 1 you add a dummy function that for now just echoes that it's giving points to the user. In a real-world application you'd include the logic for increasing the user points, say on a Redis database. Then at 2 you define the consumer callback function. The tricky bit of code at 3 is a hook to stop consuming messages. If the message body equals `quit`, then you stop the consumer. This simple trick is sure to close the channel and the connection in a clean way. Then at 4 you pass the message body to the `json_decode` function to obtain the metadata. You give `true` as the second parameter to make sure PHP will decode the JSON object as an associative array. At 5 you call the `add_points_to_user` function, passing as parameters the `user_id` that you obtained from the decoded message.

Let's test the implementation. You'll just copy the code from the publisher and modify the logic for creating the message to have a simple test script. In this case you'll take three arguments from the command line: *image ID*, *user ID*, and *image path*.



You'll encode them and send them over RabbitMQ to the consumer that you created before. We won't explain the following listing because it's the same as you saw before in listing 4.7.

Listing 4.10 Upload pictures test

```
<?php
require_once('../lib/php-amqplib/amqp.inc');
require_once('../config/config.php');

$conn = new AMQPConnection(HOST, PORT, USER, PASS, VHOST);
$channel = $conn->channel();

$channel->exchange_declare('upload-pictures',
  'fanout', false, true, false);

$metadata = json_encode(array(
      'image_id' => $argv[1],
      'user_id' => $argv[2],
      'image_path' => $argv[3]
));

$msg = new AMQPMessage($metadata, array(
      'content_type' => 'application/json',
      'delivery_mode' => 2));

$channel->basic_publish($msg, 'upload-pictures');

$channel->close();
$conn->close();
?>
```

Save this code in a file called `fanout-publisher.php` and open two terminal windows. In the first window, launch the `add-points-consumer.php` script:

```
$ php add-points-consumer.php
```

In the other window, execute the publisher, passing some random parameters to simulate a request:

```
$ php fanout-publisher.php 1 2 /path/to/pic.jpg
```

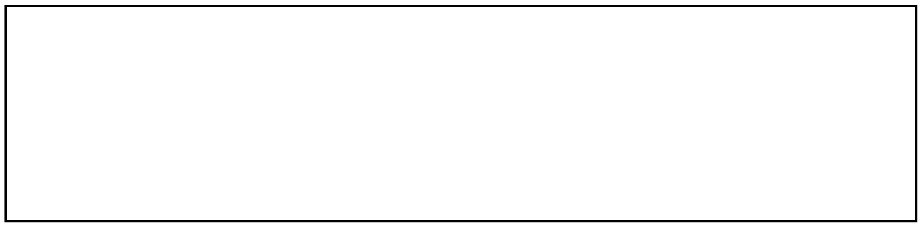
If everything went well, you can switch to the first terminal to see the following message:

```
Adding points to user: 2
```

So far nothing impressive. Let's add another consumer to see a fanout exchange and parallel processing in action. Put the code from the following listing in the file `resize-picture-consumer.php`.

Listing 4.11 Resize picture consumer

```
<?php

require_once('../lib/php-amqplib/amqp.inc');
require_once('../config/config.php');
```



```
$conn = new AMQPConnection(HOST, PORT, USER, PASS, VHOST);
$channel = $conn->channel();

$channel->exchange_declare('upload-pictures',
    'fanout', false, true, false);

$channel->queue_declare('resize-picture',
    false, true, false, false);

$channel->queue_bind('resize-picture', 'upload-pictures');

$consumer = function($msg){

    if($msg->body == 'quit'){
        $msg->delivery_info['channel']->
          basic_cancel($msg->delivery_info['consumer_tag']);
    }

    $meta = json_decode($msg->body, true);

    resize_picture($meta['image_id'], $meta['image_path']);

    $msg->delivery_info['channel']->
        basic_ack($msg->delivery_info['delivery_tag']);
};

function resize_picture($image_id, $image_path){
    echo sprintf("Resizing picture: %s %s\n",
         $image_id, $image_path);
}

$channel->basic_consume($queue,
                        $consumer_tag,
                        false,
                        false,
                        false,
                        false,
                        $consumer);

while(count($channel->callbacks)) {
    $channel->wait();
}

$channel->close();
$conn->close();
?>
```

1 Declare resize picture queue

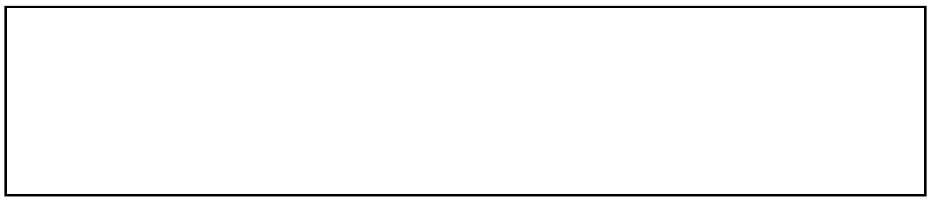
2 Bind queue to exchange

3 Resize picture

4 Resize picture function

The code in listing 4.11 is basically the same from listing 4.8. The interesting bits are at 1 and 2 where you create and bind the `resize-picture` to the `upload-picture` exchange. You can see that this uses the same exchange as the previous example. As always with AMQP, the messages are published to *one* exchange and then, depending on the bindings, they can be routed to one or several queues (or none at all).

The code continues straightforwardly; inside the consumer callback you call the `resize_picture` 3 function passing the `image_id` and `image_path` that you got from the metadata. Finally the function `resize_picture` 4 echoes a message to tell you that it's resizing the image. As before, on a real setup, here you'd want to have the code to actually resize the image.



Now, open a third window on the terminal and type

```
$ php resize-picture-consumer.php
```

Then go back to the window where you have the publisher script and run it again:

```
$ php fanout-publisher.php 1 2 /path/to/pic.jpg
```

If everything went fine, then you should see on each consumer window the following messages:

```
Adding points to user: 2
```

and

```
Resizing picture: 1 /path/to/pic.jpg
```

Based on the examples from the *add points to user* consumer, you can see that if you integrate RabbitMQ into your solution, then scaling the code to new requirements is simple. To add the *image resize* consumer you just need a function that's based on the image ID and path, and is able to load the picture from the filesystem, resize it (probably using some tool like Imagemagick), and then update the record on the database based on the image ID. The same goes for notifying the user's friends. Taking the user ID as a parameter, you can retrieve the user's contacts from the database and then send a notification, perhaps in the form of an email, to each of those friends.

What you can learn from this example is that the power of messaging with RabbitMQ resides in how you combine exchanges and queues together. If you need some way to filter out messages, then you can use a topic exchange as in the previous section. Does one action in your application trigger several others that can run in parallel? Then use topic exchanges. If you want to "spy" on a flow of messages and then quit without leaving traces, then use anonymous queues set to be autodeleted. Once you get used to thinking about messaging patterns, you'll start seeing how simple some programming tasks can become.

But the advantages of this design over the one where everything happens in the same module don't stop here. Imagine now that the pictures are being resized too slowly; you need more computing power and you don't want to change the code. That's easy to solve. You can launch more consumer processes and RabbitMQ will take care of distributing the messages accordingly. Even if they're on different machines, it's no problem. Try to imagine now how you'd scale the original code, where everything happened sequentially while you were serving the request to the user. As you saw, parallel processing with RabbitMQ is simple.

4.3 Remember me: RPC over RabbitMQ and waiting for answers

There are many ways of doing remote procedure calls (RPC)—everything from UNIX RPC to REST APIs and SOAP. What all of these traditional methods of RPC have in common is a tight linkage between the client and server. The client directly connects to

the server, makes a request, and then blocks, waiting for a response from the server. This model has a lot of benefits in that its point-to-point nature makes the topology simple at small scale. But that simple topology also limits its flexibility and increases its complexity when it becomes time to scale up. For example, how do your clients discover where to find servers with the services they want when there are multiple servers? SOAP and most enterprise RPCs have come up with complex supplementary protocols and service directories that layer on additional complexity and points of failure, all in the name of being able to serve APIs from multiple RPC servers without tight coupling between the clients and the server. Also, what happens if the RPC server your client is talking to crashes? It's up to the client to reconnect, and if the server is completely down, to rediscover a new server offering the same services—and the client still has to retry the API call once all that's done.

What if, instead of complex directories and multiple protocols, you could do RPC over one protocol? What if your client could issue an API call without worrying about which server was going to serve it, and what to do if the server failed? Using an MQ broker to do RPC can give you all of these things. When you use RabbitMQ for RPC, you're simply publishing a message. It's up to RabbitMQ to use bindings to route the message to the appropriate queue where it'll be consumed by the RPC server. RabbitMQ does all the hard work of getting the message to the right place, load balancing RPC messages across multiple RPC servers, and even retasking an RPC message to another server when the server it was assigned to crashes. All of this without complicated WS-* protocols, or any routing intelligence on the part of the client. The question is, how do you get replies back to the client? After all, your experience so far with RabbitMQ has been fire-and-forget.

4.3.1 Private queues and sending acknowledgements

Since AMQP messages are unidirectional, how can an RPC server reply back to the original client with a result? With RabbitMQ in the middle, the RPC server doesn't even know the identity of the calling client unless there's an application-specific ID in the message payload. Thankfully, the guys at RabbitMQ have an elegant solution: use messages to send replies back. On every AMQP message header is a field called `reply_to`. Within this field the producer of a message can specify the queue name they'll be listening to for a reply. The receiving RPC server can then inspect this `reply_to` field and create a new message containing the response with this queue name as the routing key.

You might be saying yourself, "That sounds like a lot of work to create a unique queue name every time. How do we keep other clients from reading the replies?" Once again, RabbitMQ rides to the rescue. You might remember from chapter 1 that if you declare a queue with no queue name, RabbitMQ will assign one for you. This name happens to be a unique queue name, and when declared with the `exclusive` parameter ensures that only you can read from the queue. All your RPC clients have to do is declare a temporary, exclusive, anonymous queue, and include the name of that

queue in the `reply_to` header of their RPC message, and the server now has a place to send the response. Note that we didn't say anything about binding the reply queue to an exchange. This is because when the RPC server publishes its reply message to RabbitMQ without an exchange specified, RabbitMQ knows that it's targeted for a reply queue and that the routing key is the queue's name.

Enough talk; let's look at how you get RPC working with RabbitMQ in real code.

4.3.2 Simple JSON RPC with reply_to

The first thing you need is an RPC server. Before we dive into the code, it might help to take a look at the flow of your RPC client and server, shown in figure 4.5.

In the following listing you'll build a simple API server that implements a ping call. This call's only function is to receive the ping invocation from the client, and send a `Pong!` reply with the timestamp included by the client in the original call.

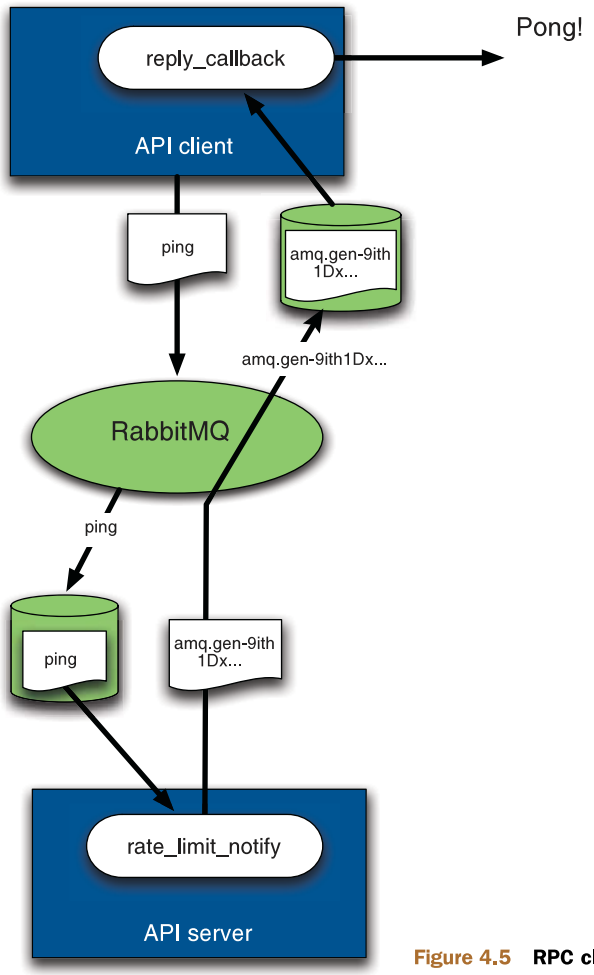


Figure 4.5 RPC client and server flow

Listing 4.12 API server—rpc_server.py

```
import pika, json

creds_broker = pika.PlainCredentials("rpc_user", "rpcme")
conn_params = pika.ConnectionParameters("localhost",
                                        virtual_host = "/",
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)
channel = conn_broker.channel()

channel.exchange_declare(exchange="rpc",
                         type="direct",
                         auto_delete=False)
channel.queue_declare(queue="ping", auto_delete=False)
channel.queue_bind(queue="ping",
                   exchange="rpc",
                   routing_key="ping")

def api_ping(channel, method, header, body):
    """'ping' API call."""
    channel.basic_ack(delivery_tag=method.delivery_tag)
    msg_dict = json.loads(body)
    print "Received API call...replying..."
    channel.basic_publish(body="Pong!" + str(msg_dict["time"]),
                          exchange="",
                          routing_key=header.reply_to)

channel.basic_consume(api_ping,
                      queue="ping",
                      consumer_tag="ping")

print "Waiting for RPC calls..."
channel.start_consuming()
```

Establish connection to broker

Declare exchange and ping call queue

Wait for RPC calls and reply

We've covered the setup and connection to RabbitMQ, so let's skip forward to the interesting part where the exchange and queues for receiving the API calls are created:

```
channel.exchange_declare(exchange="rpc",
                         type="direct",
                         auto_delete=False)
channel.queue_declare(queue="ping", auto_delete=False)
channel.queue_bind(queue="ping",
                   exchange="rpc",
                   routing_key="ping")
```

What you've done here is set up a typical direct exchange and created a queue and binding. For the API, you're following a pattern where the name of the RPC function call is what you use as the binding pattern (and queue name for those calls). In this case, the ping API call is created by binding the `ping` queue to the `rpc` exchange using `ping` as the binding pattern. All your clients need to do is put `ping` as their routing key and their arguments into the message body. You could also use more complex routing of RPC requests by using a topic exchange. Next you need to set up your consumer subscription:

```
def api_ping(channel, method, header, body):
    """'ping' API call."""
    channel.basic_ack(delivery_tag=method.delivery_tag)
    msg_dict = json.loads(body)
    print "Received API call...replying..."
    channel.basic_publish(body="Pong!" + str(msg_dict["time"]),
                          exchange="",
                          routing_key=header.reply_to)

channel.basic_consume(api_ping,
                      queue="ping",
                      consumer_tag="ping")
```

`api_ping` will now be invoked every time a message is assigned to you by RabbitMQ via the `ping` queue. All of this is similar to what you've done so far in the book. What you'll notice is different is the `basic_publish` command you issue after acknowledging the call message. Wait a minute! How are you able to publish a reply on the same channel you were consuming on? Didn't we say that was impossible?! Actually, in this case it's possible because the Pika library won't start consuming again until your `api_ping` function returns. More important to focus on is the configuration of the `basic_publish` command. It's using the `reply_to` from the header as the routing key for the reply message. Also, unlike any other publish you'll ever do with RabbitMQ, there's no exchange you're publishing to. Those are the only two special components you need to know about making RPC work over Rabbit: publish the reply using the `reply_to` as the target, and publish without an exchange specification.

How about the RPC client? What does it look like and how do you set up your reply queue? Let's take a peek at the following listing.

Listing 4.13 API client—rpc_client.py

```
import time, json, pika

creds_broker = pika.PlainCredentials("rpc_user", "rpcme")
conn_params = pika.ConnectionParameters("localhost",
                                        virtual_host = "/",
                                        credentials = creds_broker)
conn_broker = pika.BlockingConnection(conn_params)
channel = conn_broker.channel()

msg = json.dumps({"client_name": "RPC Client 1.0",
                  "time" : time.time()})
result = channel.queue_declare(exclusive=True, auto_delete=True)
msg_props = pika.BasicProperties()
msg_props.reply_to=result.method.queue

channel.basic_publish(body=msg,
                      exchange="rpc",
                      properties=msg_props,
                      routing_key="ping")

print "Sent 'ping' RPC call. Waiting for reply..."

def reply_callback(channel, method, header, body):
    """Receives RPC server replies."""
```

Establish connection to broker

Issue RPC call and wait for reply

```
    print "RPC Reply --- " + body
    channel.stop_consuming()

channel.basic_consume(reply_callback,
                      queue=result.method.queue,
                      consumer_tag=result.method.queue)

channel.start_consuming()
```

The heart of making RPC work on the client side is this bit here:

```
result = channel.queue_declare(exclusive=True, auto_delete=True)
msg_props = pika.BasicProperties()
msg_props.reply_to=result.queue
```

In those three lines you create your reply queue and set the `reply_to` header on the message to the name of the new queue. When you declare the reply queue, make sure to set `exclusive=True` and `auto_delete=True`. This ensures that no one else can pilfer your messages (though the queue name created by Rabbit *is* unique), and that when you disconnect from the queue after receiving your reply, the queue will be automatically deleted by Rabbit. All that's left is to publish the API call message and subscribe your callback function to the reply queue:

```
channel.basic_publish(body=msg,
                      exchange="rpc",
                      properties=msg_props,
                      routing_key="ping")

print "Sent 'ping' RPC call. Waiting for reply..."

def reply_callback(channel, method, header, body):
    """Receives RPC server replies."""
    print "RPC Reply --- " + body
    channel.stop_consuming()

channel.basic_consume(reply_callback,
                      queue=result.method.queue,
                      consumer_tag=result.method.queue)
```

There's nothing magical about that. Once you have the reply queue set up, you can consume from it like any other queue. Just be sure not to start consuming from the queue until after you publish your API call message. Otherwise, the channel will be in consume mode and you'll get an error when you try to publish. So what does it look like on the client and server sides when you run your RPC app?

```
Client) Sent 'ping' RPC call. Waiting for reply...
Server) Received API call...replying...
Client) RPC Reply --- Pong! (Client Name: RPC Client 1.0)
        (RPC Call Issued Time: 1288111236.43)
```

You can see that the server's reply really was based on the client's call because the timestamp included in the server's reply is the one that was in the body of the client's call message. From here you can easily extend the API by creating new queues and bindings for new API methods. The best part is there's no reason why any one RPC

server needs to respond to all of the API calls. You could easily write a new RPC server that performs image processing, for example, and run it even on a different physical box than the ping API server. Your clients won't know the difference, and you're free to scale your APIs any way you see fit. RabbitMQ does the magic of making it all act like one API fabric. No special protocols. No service directories.

4.4 Summary

In this chapter we've covered the fundamental ways of writing apps that take advantage of RabbitMQ and the messaging patterns behind them. We've discussed everything from fire-and-forget patterns, like alerting and image processing, to true bidirectional communication powering RPC APIs. With these fundamental messaging architectures under your belt you're free to start designing your own sophisticated patterns that combine the fundamentals into unique solutions that accomplish your specific goals. Now that you're starting to build RabbitMQ into the heart of your application architecture, it's time we looked at how to run RabbitMQ in resilient configurations that ensure it's always available when your apps need it.