НИИТ

Лекция №10

Взаимодействие между компонентами сложной системы

- Общая информация
- AMQP
- RabbitMQ
- Celery
- Практика

python

Общая информация

При разработке многокомпонентных систем особое внимание приходится уделять организации взаимодействия между компонентами. Примерами таких систем могут служить мессенджеры, сетевые игры, тестовые фермы. В самом простом случае, для организации обмена данными между двумя компонентами можно использовать клиент-серверную архитектура, либо технологии межпоточного и межпроцессного взаимодействия. Но для более сложных случаев этих технологий явно недостаточно. Как, например, организовать управление игровыми персонажами в сетевой игре, когда новые игроки непрерывно подключаются и отключаются, часть функционала находится на стороне клиентов, а нагрузка распределена между несколькими серверами?

Проблемы обмена данными

Перечислим основные проблемы обмена данными между узлами сложных систем, которые приходится решать разработчикам этих систем

- необходимость обеспечения отказоустойчивости;
- географическое разнесение подсистем;
- наличие узлов, взаимодействующих сразу с несколькими другими;
- архитектура клиент-сервер и технология “точка-точка” оказываются недостаточно гибкими и масштабируемыми решениями для представления множественных динамически изменяющихся связей в системе.

AMQP

Для решения вышеуказанных проблем была предложена концепция брокера сообщений – фактически отдельного компонента системы, инкапсулирующего в себе весь функционал по организации связей между остальными компонентами и управлению этими связями.

В рамках этой концепции был разработан AMQP (Advanced Message Queuing Protocol) - открытый протокол для передачи сообщений между компонентами системы.

Этот протокол позволяет не задумываться над тем, где находятся получатели сообщения, сколько их, от кого надо ждать сообщение, когда оно будет доставлено получателю. Кроме этого, AMQP снимает с разработчика еще многие рутинные задачи и позволяет заниматься непосредственно проблемой, а не вспомогательными задачами.

AMQP

AMQP имеет три базовых понятия:

- Обменник или точка обмена (exchange)
- Очередь (queue)
- Связь или маршрут (routing key)

Обмен сообщениями осуществляется в пределах одного обменника, в котором определены связи, являющиеся своеобразными маршрутами, по которым идут сообщения, попавшие в этот обменник.

Каждый маршрут связывает обменник с одной или несколькими очередями. Программное обеспечение, реализующее описанные действия, называют AMQP-сервером. Узлы, помещающие сообщения в обменники и получающие их из очередей, называются AMQP-клиентами.

Если вкратце: AMQP-сервер предоставляет шину обмена данными, а AMQP-клиенты используют эту шину для обмена сообщениями между собой.

RabbitMQ

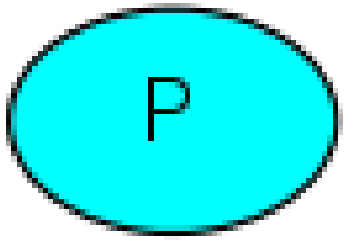
RabbitMQ — платформа, реализующая систему обмена сообщениями между компонентами программной системы (Message Oriented Middleware) на основе стандарта AMQP (Advanced Message Queuing Protocol).

RabbitMQ - это брокер сообщений. Он является open source программным обеспечением. Его основная цель – принимать и отдавать сообщения. Если проводить аналогию с доставкой почты: мы просто опускаем письмо в почтовый ящик, через почтовые отделения, почтовую службу и почтальонов оно доходит до получателя. RabbitMQ фактически берет на себя обязанности и почтового ящика, и почтовой службы, и почтальонов.

RabbitMQ: поставщик

В RabbitMQ, а также обмене сообщениями в целом, используется следующая терминология:

- Producer (поставщик) – программа, отправляющая сообщения. В схемах он будет представлен кругом с буквой «P»:

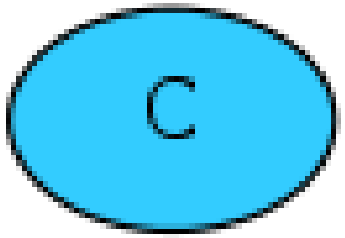


RabbitMQ: очередь

- Queue (очередь) – имя «почтового ящика». Она существует внутри RabbitMQ. Хотя сообщения проходят через RabbitMQ и приложения, хранятся они только в очередях. Очередь не имеет ограничений на количество сообщений, она может принять сколь угодно большое их количество – можно считать ее бесконечным буфером. Любое количество поставщиков может отправлять сообщения в одну очередь, также любое количество подписчиков может получать сообщения из одной очереди. В схемах очередь будет обозначена стеком и подписана именем:

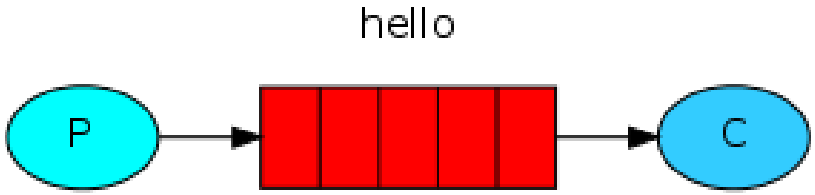
RabbitMQ: получатель

- Consumer (получатель) – программа, принимающая сообщения. Обычно получатель находится в состоянии ожидания сообщений. В схемах он будет представлен кругом с буквой «C»:



RabbitMQ: передача сообщения

Давайте просто отправим сообщение, примем его и выведем на экран. Для этого нам потребуется две программы: одна будет отправлять сообщения, другая – принимать и выводить их на экран.



RabbitMQ: установка

Для работы с RabbitMQ нужно установить соответствующую программу, написанную на языке Erlang. Для этой программы также требуется предварительно установить библиотеки Erlang – т.н. OTP (Open Telecom Platform) фреймворк.

В Linux (Ubuntu) это делается при помощи следующих команд (зависимости rabbitmq-server от OTP разрешаются и доустанавливаются автоматически).

```
$ sudo apt-get update
$ sudo apt-get install rabbitmq-server
```

В Windows сначала надо установить OTP-фреймворк (соответствующая версия 64-bit или 32-bit скачивается с http://www.erlang.org/downloads).

Затем устаналивается собственно rabbitmq-server (инсталлятор скачивается с https://www.rabbitmq.com/install-windows.html).

Ну и наконец в обоих системах надо установить библиотеку Python 3 для работы с rabbitmq – pika (желательно, в виртуальном окружении).

```
$ pip3 install pika --user
```

RabbitMQ: код получателя

```
# файл receive.py
import pika

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

channel.queue_declare(queue='hello')

def callback(ch, method, properties, body):
    print('Received %r'.format(body))

channel.basic_consume(callback, queue='hello', no_ack=True)

print('Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

RabbitMQ: код поставщика

```
# файл send.py
import pika

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

channel.queue_declare(queue='hello')

channel.basic_publish(exchange='', routing_key='hello', body='Hello World!')

print("Sent 'Hello World!'")
connection.close()
```

Запускаем сначала receive.py, затем send.py (можно несколько раз) – видим, как получатель выводит на экран сообщения от поставщиков.

RabbitMQ: асинхронная обработка запросов с распределением нагрузки

Предыдущий пример не показывает никаких особых преимуществ использования rabbitmq – все то же можно реализовать с помощью средств встроенной библиотеки multithreading.

Усложним задачу: создадим очередь, которая будет использоваться для распределения ресурсоемких задач между несколькими подписчиками.

Основная цель такой очереди – асинхронная обработка запросов, т.е. не начинать выполнение задачи прямо сейчас и не ждать, пока оно завершится. Вместо этого задачи откладываются. Каждое сообщение соответствует одной задаче. Программа-обработчик, работающая в фоновом режиме, примет задачу на обработку, и через какое-то время она будет выполнена. При запуске нескольких обработчиков задачи будут разделены между ними.

Такой принцип работы особенно полезен для применения в веб-приложениях, где невозможно обработать ресурсоемкую задачу сразу при поступлении запроса.

RabbitMQ: код получателя

Усложним код receiver.py, дополнив вызываемую функцию callback print'ом о старте с выводом пришедшего сообщения, time.sleep(random.randint(1, 5)) и print'ом о завершении обработки.

```
# файл receive.py
import pika

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

channel.queue_declare(queue='hello')

def callback(ch, method, properties, body):
    print('Start processing {}'.format(body.decode()))
    t = time.time()
    time.sleep(random.randint(1, 5))
    print('Processed for {:3.2f} sec.'.format(time.time() - t))

channel.basic_consume(callback, queue='hello', no_ack=True)

print('Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

RabbitMQ: код поставщика

Усложним код send.py, отправляя данные несколько раз в цикле (данные содержат номер текущей итерации).

```
# файл send.py
import pika

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

channel.queue_declare(queue='hello')

for i in range(5):
    channel.basic_publish(exchange='',
                          routing_key='hello',
                          body=('Hello World {}'.format(i)).encode())

print("Sent 'Hello World!'")
connection.close()
```

Запускаем несколько receive.py, затем send.py – видим, как RabbitMQ передает каждое новое сообщение следующему подписчику, равномерно распределяя нагрузку между обработчиками сообщений.

RabbitMQ: подтверждение сообщений

Что произойдет, если один из получателей отключится после получения сообщения, так его и не обработав? Сообщение так и останется необработанным. Чтоб предотвратить такой исход, RabbitMQ поддерживает хранение сообщений до подтверждения их обработки. Подтверждение (ack) отправляется подписчиком для информирования RabbitMQ о том, что полученное сообщение было обработано и RabbitMQ может его удалить.

RabbitMQ: включаем механизм подтверждения сообщений

- Если получатель прекратил работу и не отправил подтверждение, RabbitMQ поймет, что сообщение не было обработано, и передаст его другому получателю. Для этого нужно передавать в basic_consume в параметр no_ack значение False вместо True. А в конце callback делать channel.basic_ack(delivery_tag = method.delivery_tag) для отправки подтверждения. Так мы ничего не потеряем, если получатель прекратил работу.
- Чтобы не потерять сообщение даже когда отключается сервер RabbitMQ, необходимо создавать устойчивую очередь, передавая в метод queue_declare параметр durable=True, как на стороне получателя, так и на стороне поставщика (если RabbitMQ после запуска предыдущих примеров не перезапускался, придется создать новую очередь).
- Также можно сделать устойчивыми сами сообщения. Для это при публикации сообщения нужно передавать параметр properties= pika.BasicProperties(delivery_mode = 2).

RabbitMQ: код получателя

Изменим код в соответствии с рекомендациями:

```
# файл receive.py
import pika

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

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

def callback(ch, method, properties, body):
    print('Start processing {}'.format(body.decode()))
    t = time.time() time.sleep(random.randint(1, 5))
    print('Processed for {:3.2f} sec.'.format(time.time() - t))
    channel.basic_ack(delivery_tag = method.delivery_tag)

channel.basic_consume(callback, queue='hello', no_ack=False)

print('Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

RabbitMQ: код поставщика

```
# файл send.py
import pika

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

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

for i in range(5):
    channel.basic_publish(exchange='',
                          routing_key='hello',
                          body=('Hello World {}'.format(i)).encode(),
                          properties=pika.BasicProperties(delivery_mode=2))

print("Sent 'Hello World!'")
connection.close()
```

RabbitMQ: проверяем механизм подтверждения сообщений

Выполним проверку: запустим последовательно receive.py и send.py, отправим сообщения и прервем на середине их обработку, выключив receive.py. Затем запустим receive.py заново. Обратите внимание, с какого сообщения продолжит обработку receive.py.
Примечание: для перезапуска rabbitmq-server (чтоб использовать тот же самый queue_name) в Ubuntu надо выполнить команды:

```
$ invoke-rc.d rabbitmq-server stop
$ invoke-rc.d rabbitmq-server start
```

В Windows достаточно кликнуть по ярлыкам, ссылающимся на скриптовый файл rabbitmq-service.bat, запускаемый с соответствующими параметрами:
RabbitMQ Service – stop
RabbitMQ Service – start

RabbitMQ: равномерное распределение сообщений

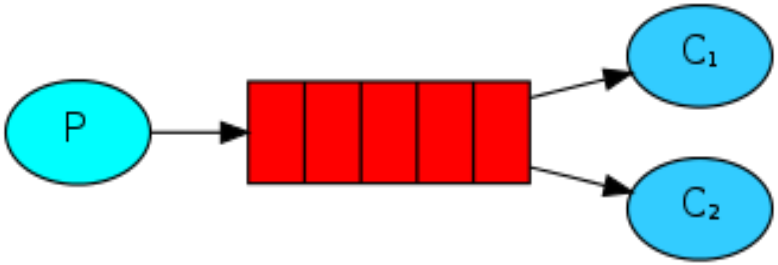
Получившийся код все равно имеет один недостаток. Пока что RabbitMQ передает сообщения получателям по очереди, не учитывая количество неподтвержденных сообщений у получателей. В результате, если какое-то сообщение требует большего, чем остальные, времени на обработку, получатель, которому оно достанется будет наравне с остальными получателями принимать и другие сообщения, что является неэффективным решением.

Чтобы изменить такое поведение, мы можем использовать метод basic_qos с опцией prefetch_count=1 перед вызовом basic_consume. Это заставит RabbitMQ не отдавать получателю единовременно более одного сообщения. Другими словами, получатель не получит новое сообщение, до тех пор пока не обработает и не подтвердит предыдущее. RabbitMQ передаст сообщение первому освободившемуся получателю.

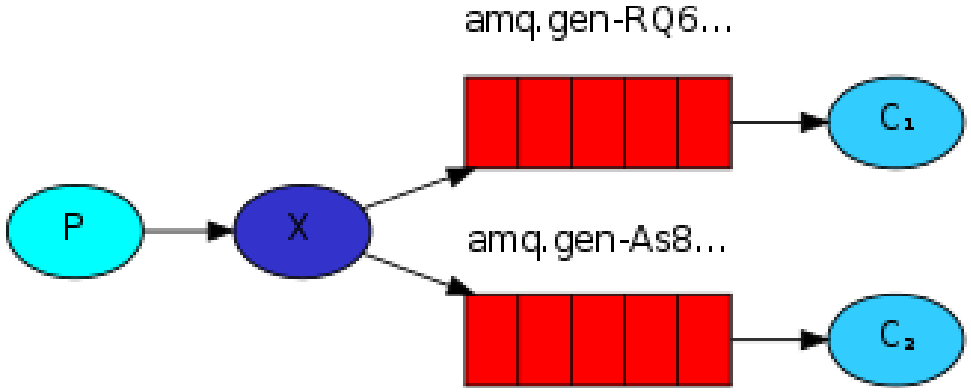
Важное замечание: если все получатели заняты, то размер очереди может увеличиваться. Следует обращать на это внимание и, при необходимости, увеличивать количество получателей.

RabbitMQ: обменник (exchange)

В предыдущих примерах была рассмотрена обработка сообщений при помощи очереди – queue. При этом каждое сообщение обрабатывалось каким-то одним получателем:



Если же требуется отправить одно сообщение нескольким различным получателям, то используется обменник (exchange):



python

RabbitMQ: обменник (exchange)

Модель отправки сообщений через обменник соответствует паттерну проектирования Издатель-Подписчик (Publisher-Subscriber), смысл которого заключается в максимальной независимости кода отправляющего сообщение от кода его обрабатывающего.

Основная идея в модели отправки сообщений через обменник состоит в том, что поставщик (producer) никогда не отправляет сообщения напрямую в очередь (довольно часто поставщик даже не знает, дошло ли его сообщение до конкретной очереди). Вместо этого поставщик отправляет сообщение в точку доступа – обменник, который точно знает, что делать с поступившими сообщениями: отправить сообщение в конкретную очередь, либо в несколько очередей, либо не отправлять никому и удалить его. Эти правила определяются типом обменника (exchange type).

RabbitMQ: exchange types

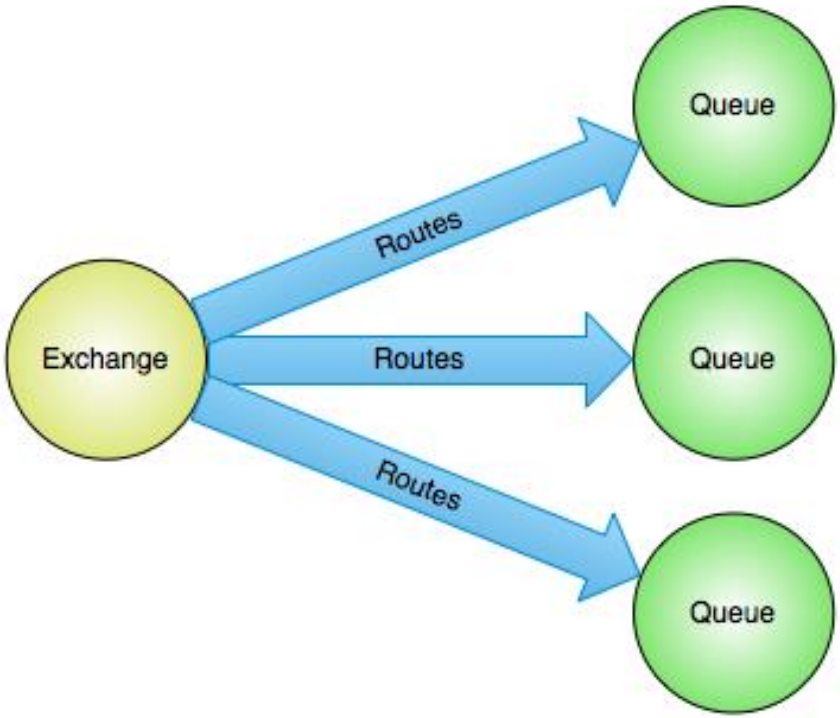
Существуют несколько типов обменников:

fanout — сообщение передаётся во все прикрепленные к точке доступа очереди;

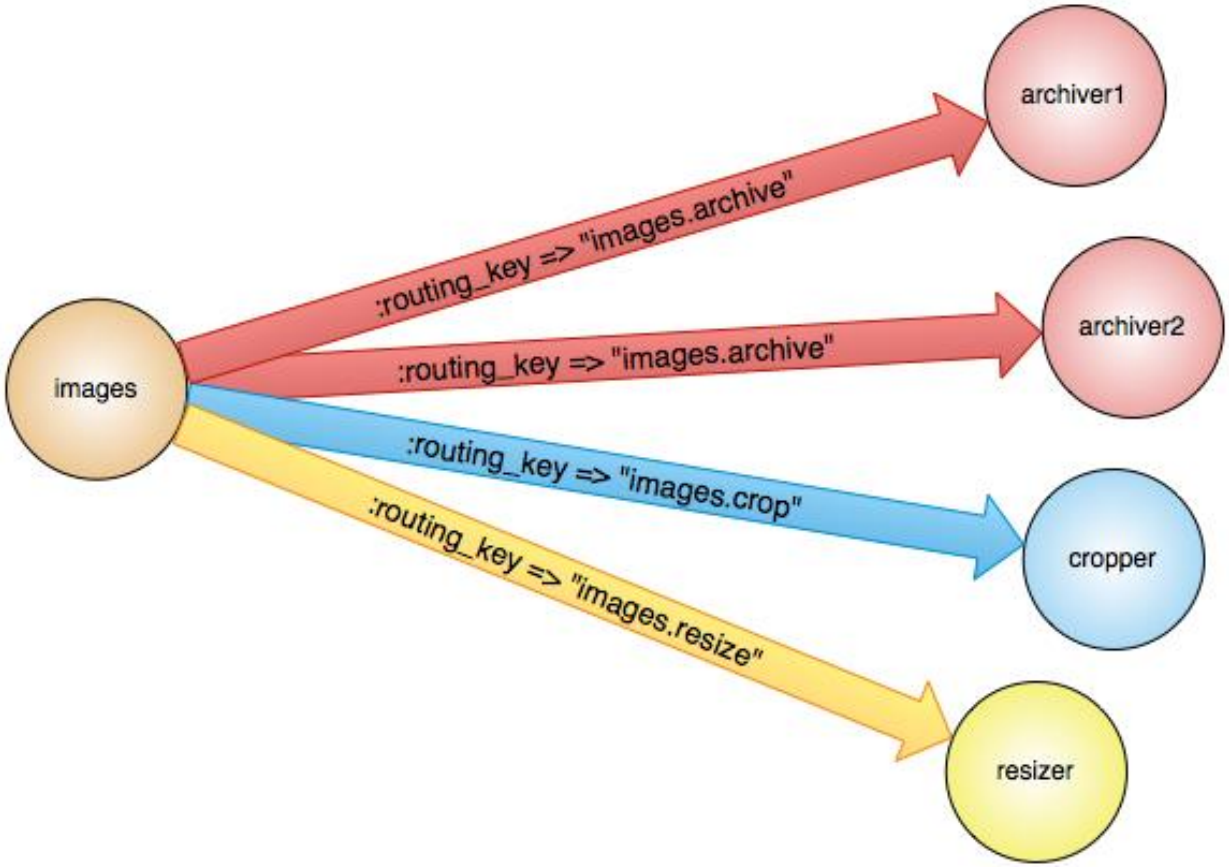
direct — сообщение передаётся в очередь с именем, совпадающим с ключом маршрутизации (routing key) (ключ маршрутизации указывается при отправке сообщения);

topic — нечто среднее между fanout и direct - сообщение передаётся в очереди, для которых совпадает маска на ключ маршрутизации, например, app.notification.sms.# — в очередь будут доставлены все сообщения, отправленные с ключами, начинающимися на app.notification.sms.

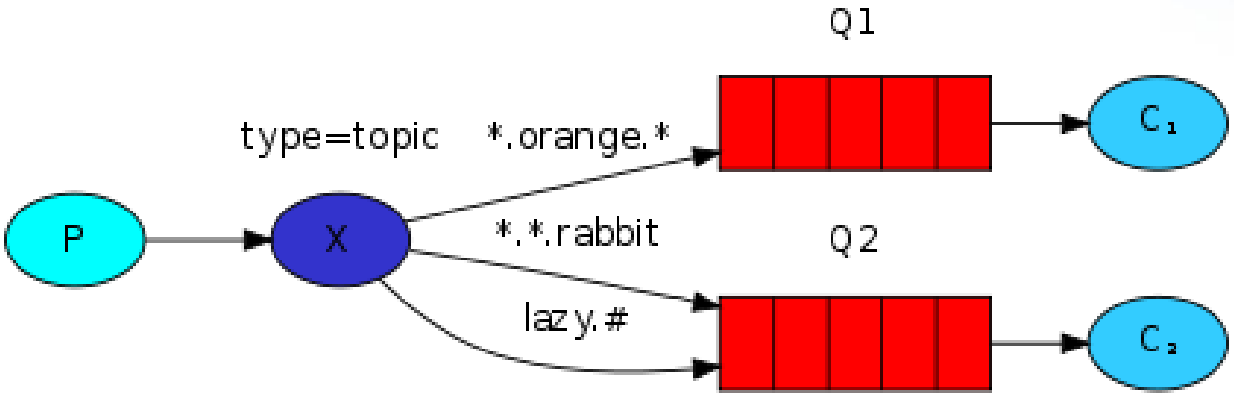
RabbitMQ: fanout exchange type



RabbitMQ: direct exchange type



RabbitMQ: topic exchange type



RabbitMQ: код подписчика

Рассмотрим пример с использованием типа обменника fanout:

```
import pika
import random

NAME = 'Logger ' + str(random.randint(1, 1000))
connection = pika.BlockingConnection(
        pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.exchange_declare(exchange='logs', exchange_type='fanout')
result = channel.queue_declare(exclusive=True)
queue_name = result.method.queue
channel.queue_bind(exchange='logs', queue=queue_name)

def callback(ch, method, properties, body):
    print('Processing msg by {}: {}'.format(body.decode(), NAME))
    channel.basic_ack(delivery_tag = method.delivery_tag)

channel.basic_qos(prefetch_count=1)
channel.basic_consume(callback, queue=queue_name, no_ack=False)

print('Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

RabbitMQ: код издателя

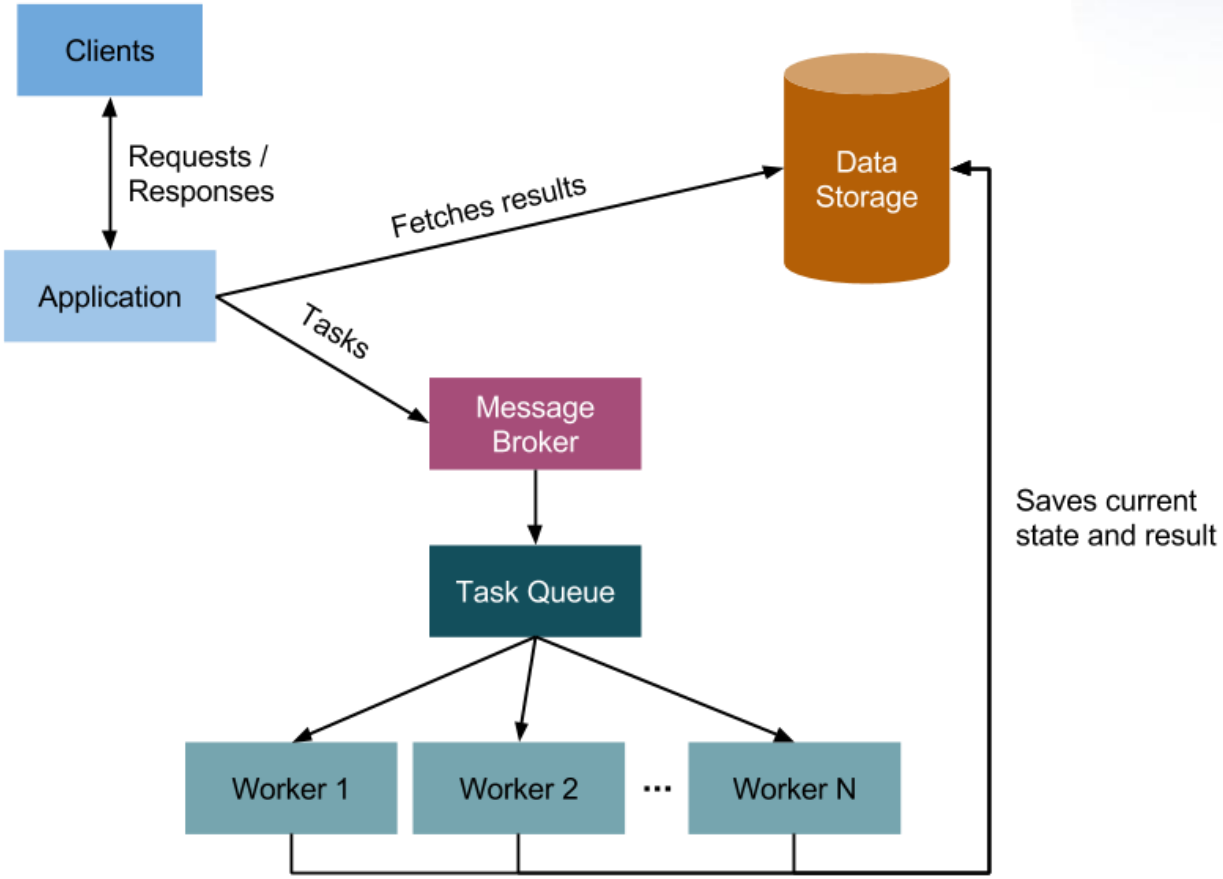
```
import pika

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

channel = connection.channel()

channel.exchange_declare(exchange='logs', exchange_type='fanout')

for i in range(5):
    channel.basic_publish(exchange='logs',
                          routing_key='',
                          body=('Hello World {}'.format(i)).encode())

print("Sent 'Hello World!'")
connection.close()
```

Celery

В Python есть библиотека Celery, предоставляющая интерфейс к распределенной асинхронной очереди заданий с широким функционалом.

Очереди заданий, как мы выяснили ранее, могут использоваться как механизм распределения работы между потоками или даже серверами. На вход очереди заданий передается единица работы – собственно, задание. Обработчик (worker) осуществляет непрерывный мониторинг очереди в поиске нового задания. Celery может состоять из множества обработчиков и брокеров сообщений, предоставляя широкие возможности для устойчивости и масштабирования системы программ.

Celery


Clients
Requests / Responses
Application
Fetches results
Data Storage
Tasks
Message Broker
Task Queue
Worker 1
Worker 2
...
Worker N
Saves current state and result

Celery

Многим разработчикам часто приходилось сталкиваться с типовыми задачами в веб-приложениях вроде отправки электронного письма посетителю или обработки загруженных данных. Чаще всего такого рода манипуляции можно выполнять в фоновом режиме. И очень часто используется связка Celery + Django.

Фоновые задачи зачастую трудоемки и склонны к сбою, главным образом, из-за внешних зависимостей. Некоторые распространенные сценарии среди сложных веб-приложений включают:

- Отправка уведомлений о подтверждении или рассылка сообщений
- Ежедневное сканирование и скрапинг некоторой информации из разных источников и сохранение этой информации
- Анализ данных
- Удаление ненужных ресурсов
- Экспорт документов/фотографий в различных форматах

Celery: запуск обработчиков

```python
import time
from celery import Celery

app = Celery('tasks', broker='amqp://guest@localhost//')
app.conf.CELERY_RESULT_BACKEND = 'amqp://'

@app.task
def add(x,y):
    print('add({0}, {1})'.format(x, y))
    time.sleep(0.1)
    return x + y

@app.task
def mul(x,y):
    print('mul({0}, {1})'.format(x, y))
    return x * y

@app.task
def xsum(l):
    print('xsum({0})'.format(l))
    return sum(l)

@app.task
def broken_task():
    raise Exception('failed')

@app.task
def on_task_error(task_id, task_name):
    print('ERROR IN TASK {0}: {1}'.format(task_id, task_name))

@app.task
def print_task(x):
    print(x)

@app.task
def on_task_success(task_id, task_name):
    print('TASK SUCCEDED {0}: {1}'.format(task_id, task_name))


if __name__ == '__main__':
    app.worker_main()
```

Celery: отправка задач

```
from celery import group, chain, chord
from tasks import (add, mul, xsum, broken_task, on_task_error, print_task,
on_task_success)

result = add.delay(1, 1) # асинхронный вызов, эквивалентен add.apply_async((1, 1))
print(result.state) # PENDING
print(result.ready()) # False
print(result.get(timeout=3)) # ожидаем результата и выводим его
print(result.state) # SUCCESS
print(result.ready()) # True

print(add(1, 1)) # синхронный вызов

print(group(add.s(i, i) for i in range(10))().get()) # запуск группы задач

# результат выполнения первой задачи используется как аргумент для следующей
print(chain(add.s(4, 4) | mul.s(8))().get())

# результат выполнения группы задач используется как аргумент для xsum,
# вызываемой как callback
print(chord(group(add.s(i, i) for i in range(10)), xsum.s())().get())

# линкование одной задачи к другой в случае неудачного выполнения первой
broken_task.apply_async(link_error=on_task_error.s('broken_task'))

# линкование одной задачи к другой в случае успешного выполнения первой
print_task.apply_async(('something', ), link=on_task_success.s('print_task'))
```

Практика

1. Написать генератор сообщений 2 типов и сделать 2-х клиентов, подписавшихся на какой-либо тип, принимающих сообщения и печатающих их в лог-файл. Для организации взаимодействия компонентов использовать RabbitMQ.
2. * Самостоятельно изучить часть функционала библиотеки Celery и написать программу, которая выводит текущую погоду на экран или записывает в файл. При этом она должна обновлять информацию о текущей погоде каждые 20 минут. Сделать это необходимо с помощью периодической таски из Celery. Информацию о погоде можно брать любым способом (например, с соответствующего сайта, используя библиотеки urllib и re).