Министерство науки и высшего образования Российской Федерации

федеральное государственное бюджетное образовательное учреждение
высшего образования
«Курганский государственный университет»

Кафедра «Безопасность информационных и автоматизированных систем»

Система обмена сообщениями RabbitMQ

Методические указания к выполнению лабораторных работ
по дисциплине «Технология построения защищённых
распределённых приложений»

для студентов, обучающихся по программе специалитета
10.05.03 «Информационная безопасность автоматизированных систем»

Курган 2024

Кафедра: «Безопасность информационных и автоматизированных систем».

Дисциплина: «Технологии построения распределенных защищенных приложений» (10.05.03 – Информационная безопасность автоматизированных систем).

Составил: канд. техн. наук, доцент Д. И. Дик.

Печатается в соответствии с планом издания, утвержденным методическим советом университета «28» декабря 2022 г.

Утверждены на заседании кафедры 18 ноября 2023 г.

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

Цель лабораторной работы заключается в закреплении теоретических основ курса и получении первоначальных навыков использования системы обмена сообщениями RabbitMQ.

2 ОБЩИЕ СВЕДЕНИЯ

2.1 Advanced Message Queuing Protocol

AMQP (Advanced Message Queuing Protocol) — открытый протокол для передачи сообщений.

Протокол AMQP определяет три понятия:

− exchange (обменник, или точка обмена) — принимает и распределяет сообщения по очередям. Пересылка сообщения в очередь осуществляется на основе привязок (binding);

− queue (очередь) — упорядоченный набор сообщений, из которого сообщения доставляются потребителям (consumers) в порядке FIFO. Одна очередь может использоваться несколькими потребителями;

− binding (привязка) устанавливает связь между точкой обмена и очередью и представляет собой правило, определяющее, какие сообщения должны размещаться в очереди. Может существовать несколько привязок, связывающих точку обмена с очередью.

2.2 Брокер сообщений RabbitMQ

RabbitMQ – это реализующий протокол AMQP, брокер сообщений с открытым исходным кодом. RabbitMQ написан на языке программирования Erlang.

Схему работы RabbitMQ можно представить следующим образом:

а) издатель (producer) генерирует сообщение и отправляет его определенной точке обмена;

б) сообщение сохраняется в оперативной памяти или на диске;

в) точка обмена в соответствии с правилами привязки передает сообщение его в одну или несколько очередей, в которые размещается ссылка на сообщение;

г) как только потребитель (consumer) будет готов получить сообщение из очереди, брокер создает копию сообщения и отправляет ее потребителю;

д) потребитель подтверждает брокеру получение сообщения;

е) после получения подтверждения брокер удаляет ссылку на сообщение из очереди и, если сообщение доставлено, из всех очередей удаляет само сообщение.

2.3 Подключение и каналы

Обмен между клиентом и сервером осуществляется посредством каналов, создаваемых в рамках подключения. При этом для одного подключения может создаваться несколько каналов. Для обращения к каналу используется синхронный удаленный вызов процедур, поэтому для отправки параллельных запросов к серверу необходимо открывать несколько каналов. Для каждого канала на сервере создается новый Erlang процесс. Соответственно с увеличением числа каналов растет потребление памяти.

Соединение и каналы должны быть постоянными, поскольку их создание связано со значительными затратами.

2.4 Точки обмена

Перед публикацией сообщений должна быть создана точка обмена. Сообщение, публикуемое в несуществующую точку обмена, удаляется.

Точка обмена осуществляет маршрутизацию сообщений между очередями на основе привязок, представляет собой ссылку на модуль с логикой маршрутизации в базе данных mnesia и не порождает отдельного Erlang-процесса, что делает ее создание очень экономичным (для создания тысячи точек обмена требуется порядка 1 МБ памяти).

2.5 Типы точек обмена

В зависимости от способа реализации привязки в RabbitMQ можно выделить несколько типов точек обмена.

Direct exchange — распределение сообщений по очередям осуществляется на основе сравнения ключей маршрутизации очереди и сообщения. Сообщение пересылается в очереди, ключ маршрутизации которого в точности совпадает с ключом маршрутизации, передаваемым вместе с публикуемым сообщением (рисунок 1).

Также возможна маршрутизация по умолчанию, когда сообщение пересылается в очередь, имя которой соответствует ключу маршрутизации.

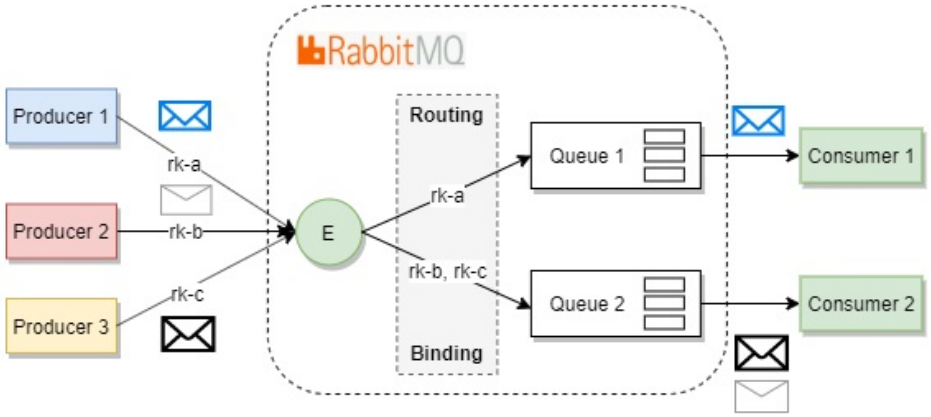

Рисунок 1 – Точка обмена Direct exchange

Topic exchange – распределение сообщений по очередям осуществляется на основе сравнения ключа маршрутизации сообщения с заданным для очереди шаблоном. Шаблон представляет собой разделенные точками слова (построенные из букв AZ, az и цифр 0-9), часть из которых может заменяться символами маски * и #. Символ * означает, что на этом месте должно находиться любое одно слово. Символ # подразумевает наличие 0 или более слов. Использующие * шаблоны работают намного быстрее, чем шаблоны, использующие #. Topic exchange работает медленнее direct exchange (рисунок 2).

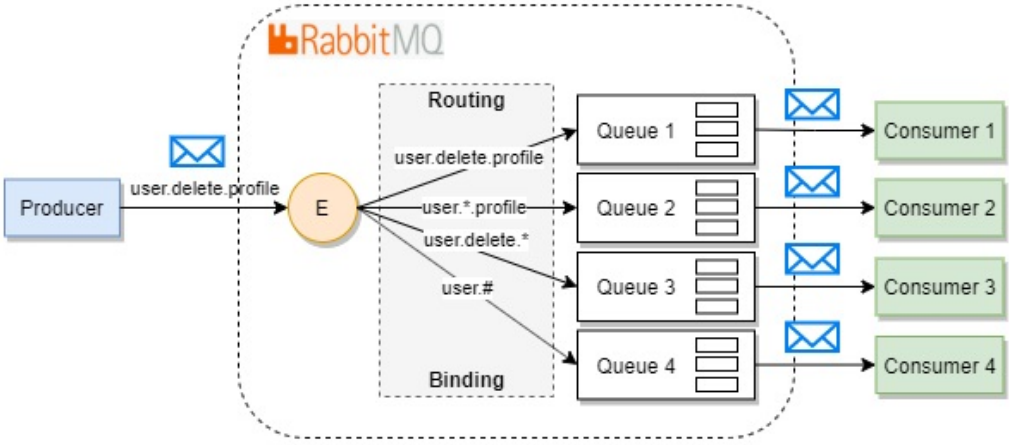

Рисунок 2 – Точка обмена Topic exchange

Fanout exchange – все сообщения доставляются во все очереди вне зависимости от наличия ключа маршрутизации. Это самый быстрый вариант привязки (рисунок 3).

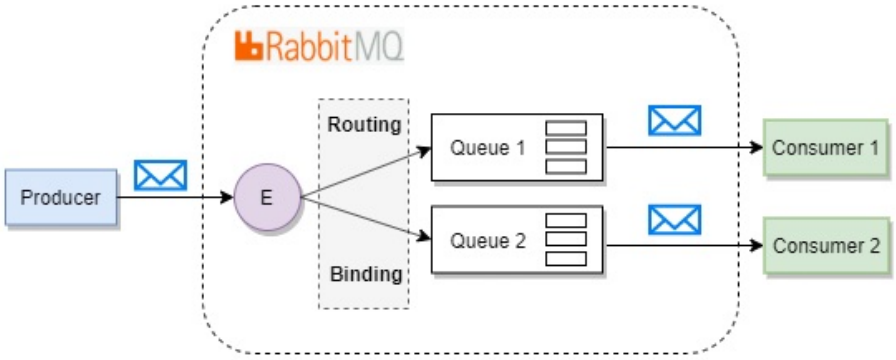

Рисунок 3 – Точка обмена Fanout exchange

Headers exchange — распределение сообщений по очередям осуществляется на основе сравнения пар (ключ, значение) свойства headers привязки с аналогичными свойствами сообщении (рисунок 4).

В headers присутствует специальный ключ x-match, который может принимать значения:

− all (по умолчанию) — в этом случае сообщение маршрутизируется при совпадении всех пар, т. е. реализуется оператор and;

− any — в этом случае сообщение маршрутизируется при совпадении каких-либо пар, т. е. реализуется оператор or.

Headers exchange обеспечивает высокую гибкость привязки, но имеет высокие накладные расходы (самый медленный вариант привязки).

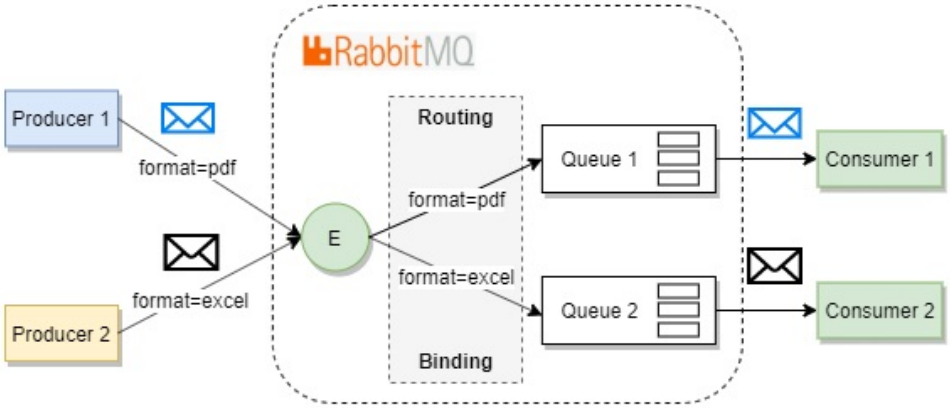

Рисунок 4 – Точка обмена Headers exchange

Consistent-hashing exchange (реализуется дополнительным плагином к RabbitMQ) — распределение сообщений по очередям осуществляется на основе веса очереди. Используется для балансировки нагрузки между очередями, очереди с одинаковым весом получат примерно одинаковое количество сообщений. Однако нужно учитывать, что используемая схема распределения основана на вычислении хэша от ключа, а веса задают долю попадающих на очередь значений из хэш пространства. Таким образом, сообщения с одинаковыми ключами всегда попадают в одну очередь, и если ключи не являются случайными,

то распределение между очередями может сильно отличаться от ожидаемого (рисунок 5).

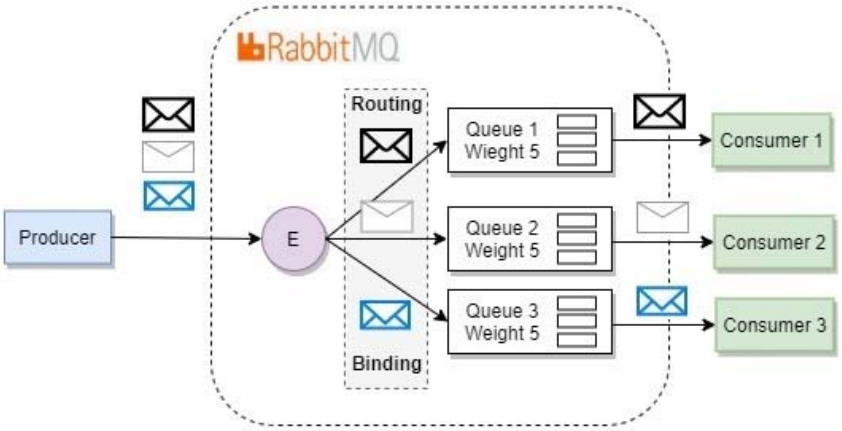

Рисунок 5 – Точка обмена Consistent-hashing exchange

RabbitMQ позволяет осуществлять привязку не только к очередям, но и к другим точкам обмена (привязка Exchange-to-Exchange), что помогает реализовывать их комбинирование (комбинирование не является частью спецификации AMQP) (рисунок 6).

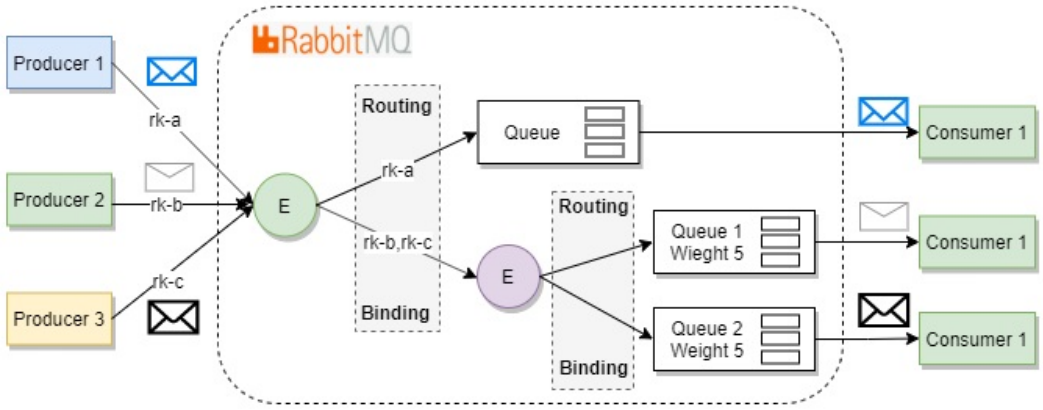

Рисунок 6 – Комбинирование точек обмена (Exchange-to-Exchange)

2.6 Очереди сообщений

Очередь (queue) — структура данных на диске или в оперативной памяти, хранящая ссылки на сообщения для последующей отдачи их копий потребителям (consumers). Очередь реализуется Erlang-процессом. Одна тысяча очередей ориентировочно занимает порядка 80 Мб оперативной памяти.

Создание очередей и привязок происходит при помощи синхронного RPC запроса из программы к серверу через предоставляемое сервером API либо через Web интерфейс администрирования.

2.6.1 Временные очереди

Существует два варианта временных очередей:

− очереди, созданные с установленным параметром autoDelete, автоматически удаляются при отсоединении от них всех клиентов;

− очереди, созданные с установленным параметром exclusive, разрешают подключаться к ним только одному потребителю и существуют, пока не закроется канал; при этом потребитель может многократно подключаться/отключаться в рамках соединения (для таких очередей параметр autoDelete эффекта не имеет).

Необходимо учитывать, что для временных очередей при разрыве связи будут потеряны все сообщения, не дошедшие до потребителя.

2.6.2 Постоянные очереди

Очереди, созданные с установленным параметром durable, являются постоянными и существуют до момента их явного удаления. Такие очереди сохраняют свое состояние и восстанавливаются после перезапуска брокера.

2.6.3 Очереди высокой доступности

При развертывании RabbitMQ в виде кластера возможно создание очередей высокой доступности (Highly Available, HA). При развертывании кластера информация о точках обмена, привязках и очередях копируется на узлы кластера. При публикации сообщения в HA очередь сообщения будет храниться на каждом узле, относящемся к данной очереди (очередь может размещаться как на всех, так и на определенных узлах кластера, обычно используют 2–3 узла). После получения сообщения потребителем оно удаляется со всех кластера.

3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

3.1 Установка операционной системы

В качестве операционной системы для нашего брокера будем использовать Ubuntu 24.04.3 LTS. Систему будем запускать под управлением среды виртуализации VirtualBox. Выставим следующие системные настройки для виртуальной машины: 20 GB пространства для жёсткого диска, два ядра и 2048 Мб памяти.

После того как была скачана и установлена операционная система, необходимо обновиться:

```
sudo apt-get update && sudo apt-get upgrade
```

Для редактирования файлов с консоли будем использовать редактор nano. Для его установки введем команду (в Ubuntu 24.04.3 LTS установлен по умолчанию):

```
sudo apt-get install nano
```

Для запуска:

```
nano файл
```

3.2 Установка RabbitMQ

Для того чтобы apt мог загружать пакеты RabbitMQ и Erlang, установим пакеты

```
sudo apt-get install curl gnupg apt-transport-https -y
```

Чтобы использовать репозиторий, содержащий последние релизы RabbitMQ, добавим ключ подписи RabbitMQ в apt-key. В результате apt будет доверять пакетам, подписанным этим ключом:

```
curl -1sLf "https://keys.openpgp.org/vks/v1/by-
fingerprint/0A9AF2115F4687BD29803A206B73A36E6026DFCA" | sudo gpg --dearmor |
sudo tee /usr/share/keyrings/com.rabbitmq.team.gpg > /dev/null

curl -1sLf https://github.com/rabbitmq/signing-
keys/releases/download/3.0/cloudsmith.rabbitmq-erlang.E495BB49CC4BBE5B.key |
sudo gpg --dearmor | sudo tee /usr/share/keyrings/rabbitmq.E495BB49CC4BBE5B.gpg
> /dev/null

curl -1sLf https://github.com/rabbitmq/signing-
keys/releases/download/3.0/cloudsmith.rabbitmq-server.9F4587F226208342.key |
sudo gpg --dearmor | sudo tee /usr/share/keyrings/rabbitmq.9F4587F226208342.gpg
> /dev/null
```

Создадим файл /etc/apt/sources.list.d/rabbitmq.list, описывающий сторонние apt репозитории RabbitMQ и Erlang:

```
sudo tee /etc/apt/sources.list.d/rabbitmq.list <<EOF

## Provides modern Erlang/OTP releases
##

deb [signed-by=/usr/share/keyrings/rabbitmq.E495BB49CC4BBE5B.gpg]
https://ppa1.novemberain.com/rabbitmq/rabbitmq-erlang/deb/ubuntu jammy main

deb-src [signed-by=/usr/share/keyrings/rabbitmq.E495BB49CC4BBE5B.gpg]
https://ppa1.novemberain.com/rabbitmq/rabbitmq-erlang/deb/ubuntu jammy main


# another mirror for redundancy

deb [signed-by=/usr/share/keyrings/rabbitmq.E495BB49CC4BBE5B.gpg]
https://ppa2.novemberain.com/rabbitmq/rabbitmq-erlang/deb/ubuntu jammy main
```

```
deb-src [signed-by=/usr/share/keyrings/rabbitmq.E495BB49CC4BBE5B.gpg]
https://ppa2.novemberain.com/rabbitmq/rabbitmq-erlang/deb/ubuntu jammy main


## Provides RabbitMQ
##
deb [signed-by=/usr/share/keyrings/rabbitmq.9F4587F226208342.gpg]
https://ppa1.novemberain.com/rabbitmq/rabbitmq-server/deb/ubuntu jammy main
deb-src [signed-by=/usr/share/keyrings/rabbitmq.9F4587F226208342.gpg]
https://ppa1.novemberain.com/rabbitmq/rabbitmq-server/deb/ubuntu jammy main


# another mirror for redundancy
deb [signed-by=/usr/share/keyrings/rabbitmq.9F4587F226208342.gpg]
https://ppa2.novemberain.com/rabbitmq/rabbitmq-server/deb/ubuntu jammy main
deb-src [signed-by=/usr/share/keyrings/rabbitmq.9F4587F226208342.gpg]
https://ppa2.novemberain.com/rabbitmq/rabbitmq-server/deb/ubuntu jammy main
EOF
```

Обновим список apt источников

```
sudo apt-get update -y
```

Установим Erlang пакеты

```
sudo apt-get install -y erlang-base \
     erlang-asn1 erlang-crypto erlang-eldap erlang-ftp erlang-inets \
     erlang-mnesia erlang-os-mon erlang-parsetools erlang-public-key \
     erlang-runtime-tools erlang-snmp erlang-ssl \
     erlang-syntax-tools erlang-tftp erlang-tools erlang-xmerl
```

Установим RabbitMQ сервер с зависимостями

```
sudo apt-get install rabbitmq-server -y --fix-missing
```

Запустим

```
sudo service rabbitmq-server start
```

Посмотрим статус запущенного сервиса

```
sudo service rabbitmq-server status
```

Проверим, что сервис запущен и консольная утилита успешно в ней взаимодействует

```
sudo rabbitmq-diagnostics ping
```

Включим Web интерфейс администрирования

```
sudo rabbitmq-plugins enable rabbitmq_management
```

Создадим нового пользователя (admin) и зададим ему пароль (12345)

```
sudo rabbitmqctl add_user admin 12345
```

Присвоим созданному пользователю (admin) права администратора

```
sudo rabbitmqctl set_user_tags admin administrator
```

Теперь Web панель администрирования доступна по адресу http://host:15672, где host – IP-адрес, или разрешаемое имя узла, на котором установлен RabbitMQ.

3.3 Установка библиотеки для отправки сообщений в RabbitMQ

Для изучения работы RabbitMQ воспользуемся библиотекой Pika для языка программирования Python. Убедимся, что Python уже установлен в системе

```
python3 --version
```

Установим PIP для Python версии 3:

```
sudo apt install python3-pip
```

Установим библиотеку Pika:

```
sudo python3 -m pip install pika --upgrade
```

3.4 Простой обмен сообщениями

С помощью редактора nano создадим файл send.py

```
nano send.py
```

со следующим содержимым

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

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

channel.queue_declare(queue='hello')

channel.basic_publish(exchange='', routing_key='hello', body='Hello World!')
print(" [x] Sent 'Hello World!'")
connection.close()
```

Программа делает следующее:

− Подключается к брокеру сообщений, находящемуся на локальном хосте. Для подключения к брокеру, находящемуся на другой машине, достаточно заменить «localhost» на IP адрес нужной машины.

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

− Создает очередь с именем «hello», в которую будет отправлено сообщение (если отправить сообщение в несуществующую очередь, RabbitMQ его проигнорирует):

```
channel.queue_declare(queue='hello')
```

− Отправляет сообщение «Hello World!» в очередь «hello» (задана в «routing_key») через точку обмена по умолчанию («exchange»):

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

− Закрывает соединение с брокером. Перед закрытием соединения очищается буфер отправки, и сообщение доставляется до RabbitMQ.

```
connection.close()
```

С помощью редактора nano создадим файл receive.py

```
nano receive.py
```

со следующим содержимым

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

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

    channel.queue_declare(queue='hello')

    def callback(ch, method, properties, body):
        print(" [x] Received %r" % body)

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

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

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

Программа делает следующее:

− Подключается к брокеру сообщений, находящемуся на локальном хосте.

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

− Создает очередь с именем «hello».

```
channel.queue_declare(queue='hello')
```

Команда queue_declare не будет создавать новую очередь, если она уже существует, поэтому сколько бы раз не была вызвана эта команда, все равно будет создана только одна очередь. Очередь создается в обоих файлах, поскольку мы не знаем, какая программа будет запущена раньше.

− Создает callback функцию которая будет принимать сообщение. При получении каждого сообщения библиотека Pika вызывает эту callback функцию. В нашем примере функция будет выводить на экран текст сообщения.

```
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
```

− Указывает, что функция callback будет получать сообщения из очереди с именем «hello» (очередь, на которую мы хотим подписаться, должна быть объявлена. Мы сделали это ранее с помощью команды queue_declare):

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

− И, наконец, запускает бесконечный процесс, который ожидает сообщения из очереди и вызывает callback функцию, когда это необходимо.

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

С помощью программы send.py отправим сообщение

```
python3 send.py
```

С помощью консольной команды

```
sudo rabbitmqctl list_queues
```

посмотрим, что очередь с именем «hello» была создана и содержит одно сообщение.

Запустим принимающую сообщения программу

```
python3 receive.py
```

и увидим, что сообщение принято.

Созданную очередь можно посмотреть через Web интерфейс администрирования (рисунок 7).

Переключимся на другой консоль (для Ubuntu Server, если используется окно VirtualBox, то можно воспользоваться клавиатурным сочетание Alt-F2 (переход на вторую консоль, возврат обратно Alt-F1)).

В новой консоли с помощью программы send.py отправим еще одно сообщение

```
python3 send.py
```

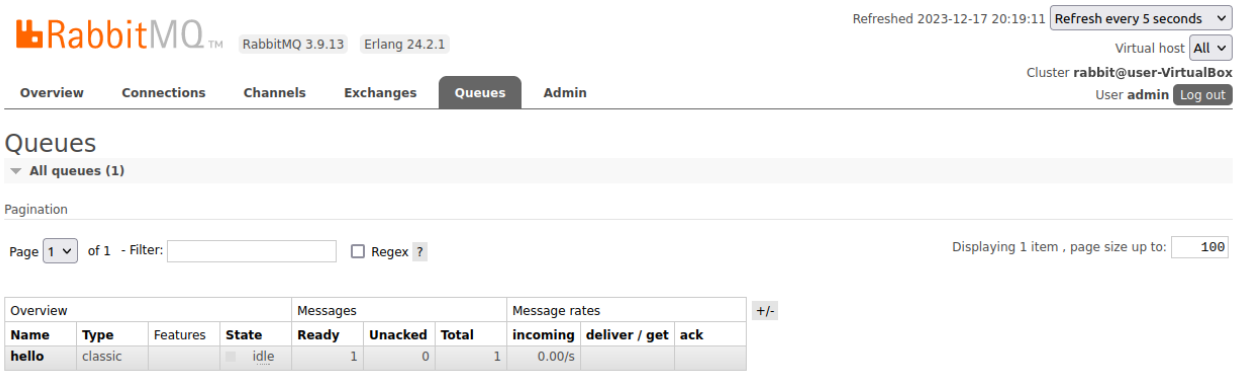

Рисунок 7 – Просмотр очереди через Web интерфейс администрирования

Вернемся обратно в исходную консоль и удостоверимся, что сообщение принято. Прервем работу программы receive.py (клавиатурным сочетанием Ctr-C).

3.5 Создание очереди задач

Скопируем программу send.py в файл с именем new_task.py

```
cp send.py new_task.py
```

Немного изменим код программы new_task.py, чтобы было возможно отправлять произвольные сообщения из командной строки. Эта программа будет отправлять сообщения в нашу очередь, планируя выполнение новых задач:

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

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

channel.queue_declare(queue='hello')

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

```
connection.close()
```

Скопируем программу receive.py в файл с именем worker.py

```
cp receive.py worker.py
```

Изменим callback функцию в программе worker.py так, чтобы симулировать выполнение полезной работы по секунде на каждую точку текста сообщения. Программа будет получать сообщение из очереди и выполнять задачу:

```
#!/usr/bin/env python
import pika, sys, os
import time

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

    channel.queue_declare(queue='hello')

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

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

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

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

Одно из преимуществ использования очереди задач – возможность выполнять работу параллельно несколькими программами. Если мы не успеваем выполнять все поступающие задачи, то можем просто прибавить количество обработчиков.

В двух консолях запустим подписчиков

```
python3 worker.py
```

В третьей консоли мы будем публиковать новые задачи. После того как подписчики запущены, можно отправлять любое количество сообщений:

```
python3 new_task.py First message.
python3 new_task.py Second message..
```

```
python3 new_task.py Third message...
python3 new_task.py Fourth message....
python3 new_task.py Fifth message.....
```

Посмотрим, что было доставлено подписчикам. Первому подписчику доставлено:

```
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received 'First message.'
 [x] Done
 [x] Received 'Third message...'
 [x] Done
 [x] Received 'Fifth message.....'
 [x] Done
```

Второму подписчику доставлено:

```
 [*] Waiting for messages. To exit press CTRL+C
 [x] Received 'Second message..'
 [x] Done
 [x] Received 'Fourth message....'
 [x] Done
```

По умолчанию RabbitMQ будет передавать каждое новое сообщение следующему подписчику. Таким образом, все подписчики получат одинаковое количество сообщений. Такой способ распределения сообщений называется циклический (алгоритм round-robin).

3.5.1 Подтверждение сообщений

В текущей реализации наших программ сообщение удаляется, как только RabbitMQ доставил его подписчику. Поэтому, если Вы остановите обработчик во время работы, задача не будет выполнена, а сообщение будет утеряно. Также будут утеряны доставленные сообщения, обработка которых еще не была начата.

Мы не хотим терять какие-либо задачи. Нам нужно, чтобы в случае аварийного выхода одного обработчика, сообщение передавалось другому.

Чтобы быть уверенным в отсутствии потерянных сообщений, RabbitMQ поддерживает подтверждение сообщений. Подтверждение (ack) отправляется подписчиком для информирования RabbitMQ о том, что полученное сообщение было обработано и RabbitMQ может его удалить.

Если подписчик прекратил работу и не отправил подтверждение, RabbitMQ поймет, что сообщение не было обработано, и передаст его другому подписчику. Так Вы можете быть уверены, что ни одно сообщение не будет потеряно, даже если выполнение программы-обработчика неожиданно прекратилось.

Для обработки сообщений отсутствует тайм-аут. RabbitMQ передаст их другому подписчику только в том случае, если соединение с первым будет закрыто, поэтому нет никаких ограничений на время обработки сообщения.

По умолчанию используется ручное подтверждение сообщений. В предыдущем примере мы принудительно включили автоматическое подтверждение сообщений, указав no_ack=True. Теперь мы уберем этот флаг и будем отправлять подтверждение из обработчика сразу после выполнения задачи.

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

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

Теперь, даже если Вы остановите обработчика, нажав Ctrl+C во время обработки сообщения, ничто не будет утеряно. После перезапуска обработчика RabbitMQ заново передаст неподтвержденные сообщения.

3.5.2 Устойчивость сообщений

Однако задачи по-прежнему будут утеряны, если прекратит работу сервер RabbitMQ.

По умолчанию при остановке или падении сервера RabbitMQ все очереди и сообщения теряются, но это поведение можно изменить. Для того чтобы сообщения оставались в очереди после перезапуска сервера, необходимо сделать как очереди, так и сообщения устойчивыми.

Сначала убедимся, что не будет потеряна очередь. Для этого необходимо объявить ее как устойчивую (durable):

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

Этот код необходимо исправить и для программы-поставщика, и для программы-подписчика. Имя очереди изменено, поскольку нельзя переопределить параметры уже существующей очереди.

Для того чтобы сообщения из очереди не терялись после перезапуска сервера RabbitMQ, необходимо пометить сообщения как устойчивые. Для этого нужно передать свойство delivery_mode со значением 2 (new_task.py):

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

3.5.3 Равномерное распределение сообщений

В рассмотренном ранее примере использовалась циклическая доставка сообщений. В результате, например, при работе двух подписчиков, если все нечетные сообщения содержат сложные задачи (требуют много времени на выполнение), а четные – простые, то первый обработчик будет постоянно занят, а второй большую часть времени будет свободен.

Так происходит, потому что RabbitMQ распределяет сообщения в тот момент, когда они попадают в очередь, и не учитывает количество неподтвержденных сообщений у подписчиков.

Для того чтобы изменить такое поведение, мы можем использовать метод basic_qos с опцией prefetch_count=1. Это заставит RabbitMQ не отдавать подписчику единовременно более одного сообщения. Другими словами, подписчик не получит новое сообщение до тех пор, пока не обработает и не подтвердит предыдущее. RabbitMQ передаст сообщение первому освободившемуся подписчику (worker.py):

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

channel.basic_qos(prefetch_count=1)
channel.basic_consume(queue='task_queue', on_message_callback=callback)
```

3.6 Публикация/Подписка

Ранее мы отправляли сообщение одному обработчику (worker). Усложним задачу – отправим сообщение одновременно нескольким подписчикам. Этот паттерн известен как «publish/subscribe» (публикация/подписка).

Чтобы понять этот шаблон, создадим простую систему логирования. Она будет состоять из двух частей: первая будет создавать логи, вторая получать их.

В нашей системе логирования каждая программа подписчик будет получать каждое сообщение. Благодаря этому, мы сможем запустить одного подписчика, например на сохранение логов на диск, а потом в любое время сможем создать другого подписчика для отображения логов на экран.

По существу, каждое сообщение будет транслироваться каждому подписчику.

3.6.1 Точки обмена (exchanges)

Основная идея в модели отправки сообщений Rabbit – поставщик (producer) никогда не отправляет сообщения напрямую в очередь. Фактически довольно часто поставщик не знает, дошло ли его сообщение до конкретной очереди.

Вместо этого поставщик отправляет сообщение в точку обмена, которая выполняет две функции:

- получает сообщения от поставщика;
- отправляет эти сообщения в очередь.

Точка обмена точно знает, что делать с поступившими сообщениями:

- отправить сообщение в конкретную очередь,
- отправить сообщение в несколько очередей,
- не отправлять никому и удалить его.

Эти правила описываются в типе точки обмена (exchange type). Существуют несколько типов: direct, topic, headers и fanout. Воспользуемся типом fanout.

Создадим файл emit_log.py:

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

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

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

message = ' '.join(sys.argv[1:]) or "info: Hello World!"
channel.basic_publish(exchange='logs', routing_key='', body=message)
print(" [x] Sent %r" % message)
connection.close()
```

Файл похож на файл new_task.py, но вместо создания очереди

```
channel.queue_declare(queue='hello')
```

мы создаем точку обмена log

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

и при публикации сообщения используем не точку обмена по умолчанию, а вновь созданную точку обмена «logs»

```
channel.basic_publish(exchange='logs', routing_key='', body=message)
```

Создание точки обмена необходимо, так как использование несуществующей точки обмена запрещено.

3.6.2 Временные очереди

Всё это время мы использовали наименование очередей («hello» или «task_queue»). Возможность давать наименования помогает указать обработчикам (workers) определенную очередь, а также делить очередь между поставщиками и подписчиками.

Но наша система логирования требует, чтобы в очередь поступали все сообщения, а не только часть. Также мы хотим, чтобы при подключении нового подписчика он видел актуальные сообщения, а не старые. Для этого нам понадобиться две вещи:

− каждый раз, когда подписчик соединяется с Rabbit, будет создаться новая очередь со случайным именем;

− каждый раз, когда подписчик отключается от Rabbit, связанная с ним очередь будет удаляться.

Для создания временной очереди подписчик создает очередь с пустым именем:

```
result = channel.queue_declare(queue='', exclusive=True)
```

Метод вернет автоматически сгенерированное имя очереди. Она может быть такой – ‘amq.gen-JzTY20BRgKO-HjmUJj0wLg.’.

Чтобы при закрытии соединения очередь автоматически удалялась, используется флаг exclusive:

```
result = channel.queue_declare(queue='', exclusive=True)
```

3.6.3 Привязка (binding)

Пока ни одна очередь не связана с точкой обмена, отправляемые в точку обмена сообщения будут теряться. Для нас это хорошо: пока нет ни одного подписчика нашей точки обмена, все сообщения могут безопасно удаляться.

Для того чтобы сказать точке обмена, чтобы она отправила сообщение в очередь, необходимо выполнить привязку (binding) между точкой обмена и очередью:

```
channel.queue_bind(exchange='logs',
                   queue=result.method.queue)
```

В результате получаем следующий код для нашего подписчика (receive_logs.py):

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

```
def main():
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()

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

    result = channel.queue_declare(queue='', exclusive=True)
    queue_name = result.method.queue

    channel.queue_bind(exchange='logs', queue=queue_name)

    print(' [*] Waiting for logs. To exit press CTRL+C')

    def callback(ch, method, properties, body):
        print(" [x] %r" % body)

    channel.basic_consume(
        queue=queue_name, on_message_callback=callback, auto_ack=True)

    channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

В двух консолях запустим подписчиков

```
python3 receive_logs.py
```

В третьей консоли мы будем публиковать логируемые сообщения. После того как подписчики запущены, можно отправлять любое количество сообщений:

```
python3 emit_log.py First log
python3 emit_log.py Second log
python3 emit_log.py Third log
```

Посмотрим, что было доставлено подписчикам. Оба подписчика получают одинаковые сообщения:

```
 [*] Waiting for logs. To exit press CTRL+C
 [x] b'First log'
 [x] b'Second log'
```

```
[x] b'Third log'
```

Не закрывая подписчиков, посмотрим, какие существуют точки обмена:

```
sudo rabbitmqctl list_exchanges
```

и видим нашу точку обмена logs, имеющую тип fanout:

```
Listing exchanges for vhost / ...
name    type
amq.fanout      fanout
amq.headers     headers
logs    fanout
amq.match       headers
amq.rabbitmq.trace      topic
        direct
amq.topic       topic
amq.direct      direct
```

Не закрывая подписчиков, также посмотрим, какие существуют привязки:

```
sudo rabbitmqctl list_bindings
```

и видим, что к точке обмена logs привязаны две очереди со случайными именами:

```
Listing bindings for vhost /...
source_name source_kind destination_name destination_kind routing_key arguments
     exchange   amq.gen-Wv_liykydPAEZACB6iFLhw   queue   amq.gen-Wv_liykydPAEZACB6iFLhw   []
     exchange   hello   queue   hello   []
     exchange   amq.gen-Fgu33oGSbRBVw328XR1vxA   queue   amq.gen-Fgu33oGSbRBVw328XR1vxA   []
logs exchange   amq.gen-Fgu33oGSbRBVw328XR1vxA   queue   amq.gen-Fgu33oGSbRBVw328XR1vxA   []
logs exchange   amq.gen-Wv_liykydPAEZACB6iFLhw   queue   amq.gen-Wv_liykydPAEZACB6iFLhw   []
```

Можно посмотреть точки обмена и очереди через графический интерфейс (рисунки 8–10). После закрытия подписчиков удостоверимся, что очереди удалились.

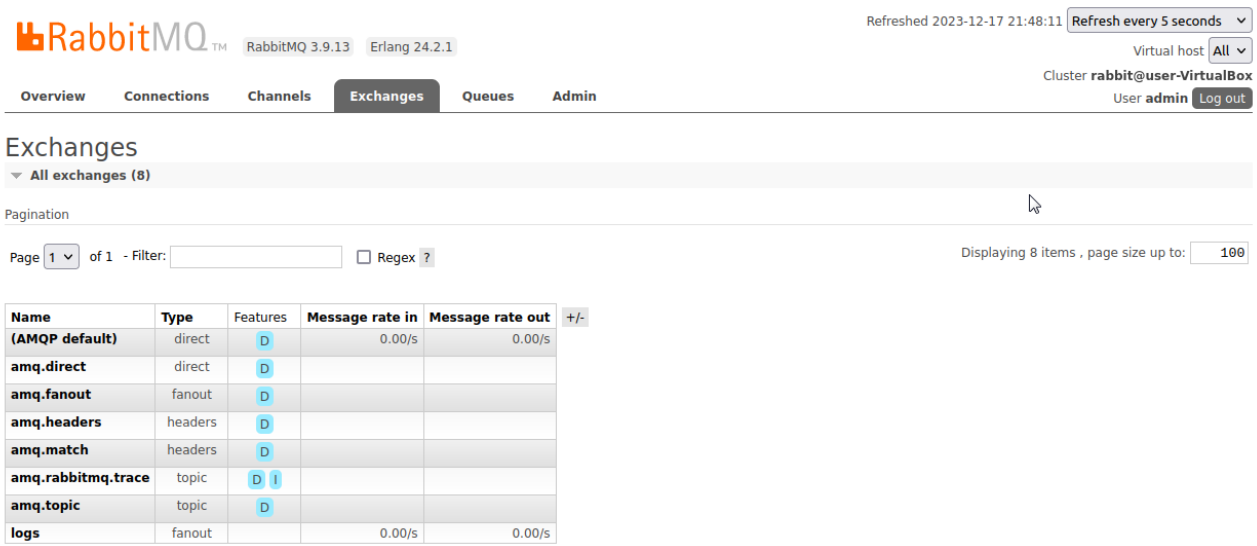

Рисунок 8 – Просмотр точек обмена через Web интерфейс администрирования

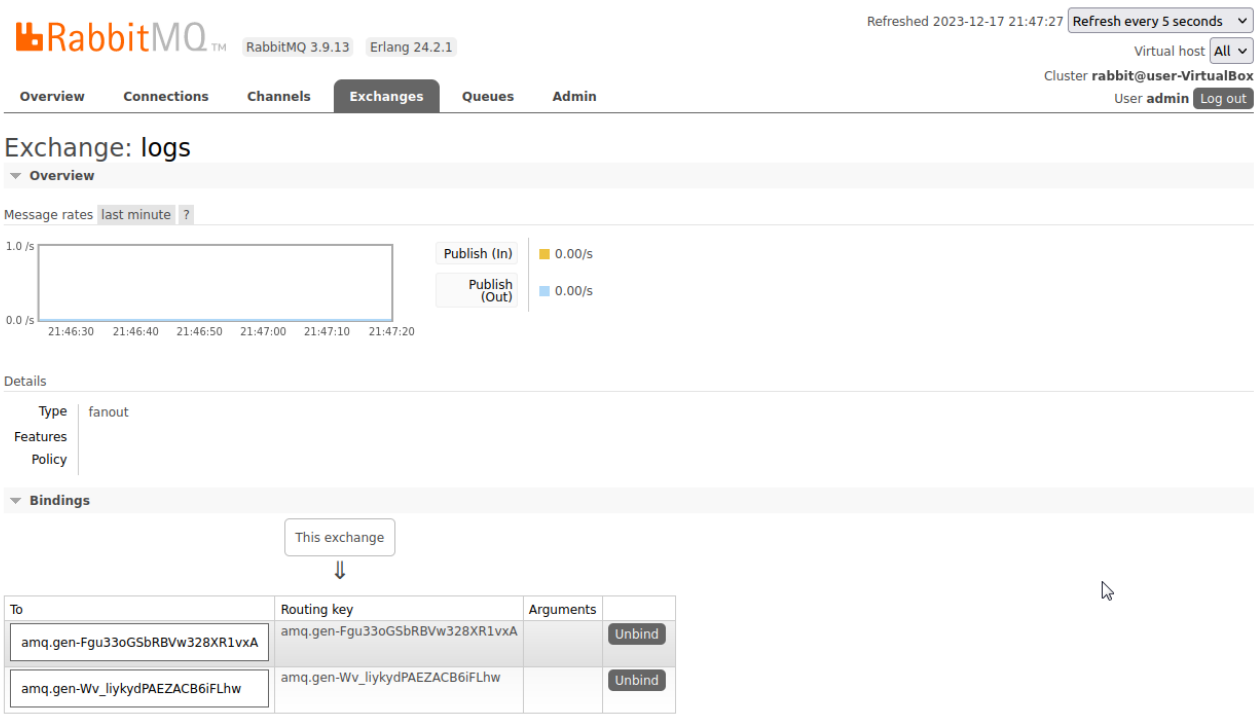

Рисунок 9 – Информация о точке обмена logs в Web интерфейсе

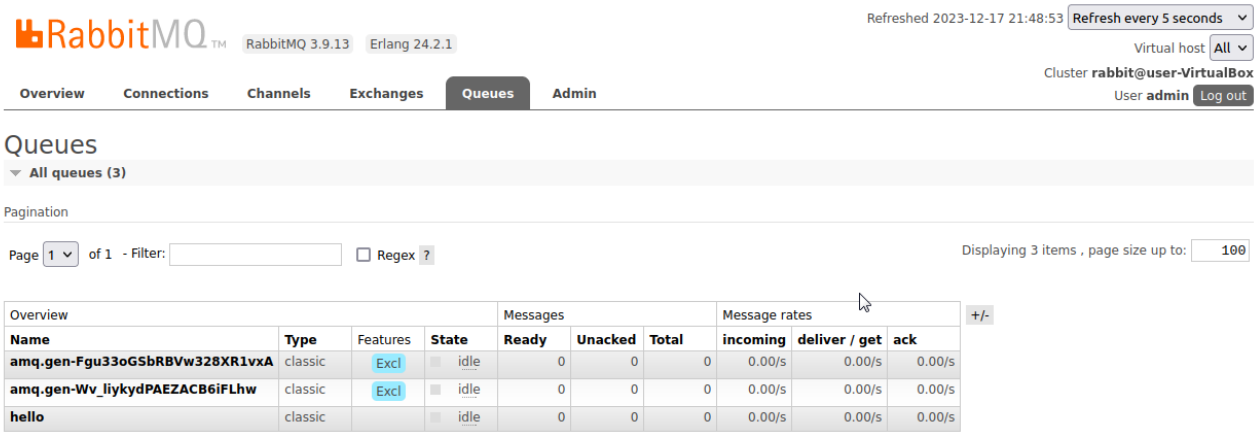

Рисунок 10 – Просмотр очередей через Web интерфейс администрирования

3.7 Маршрутизация сообщений

Наша система логирования в предыдущей статье отправляла всем подписчиками все сообщения. Расширим систему так, чтобы фильтровать сообщения по степени критичности.

Вместо точки обмена с типом fanout, которая выполняла простую трансляцию сообщений, будем использовать тип direct. Его алгоритм очень прост — сообщения идут в ту очередь, binding_key которой совпадает с routing key сообщения.

Создадим программу для отправки логируемых событий emit_log_direct.py:

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

```
import pika
import sys

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

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

severity = sys.argv[1] if len(sys.argv) > 1 else 'info'
message = ' '.join(sys.argv[2:]) or 'Hello World!'
channel.basic_publish(
    exchange='direct_logs', routing_key=severity, body=message)
print(" [x] Sent %r:%r" % (severity, message))
connection.close()
```

Файл похож на файл emit_log.py, но для точки обмена используется тип direct, а точка обмена имеет теперь имя direct_logs

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

Также программа теперь принимает два параметра. Первый задает степень критичности события, второй – само логируемое событие.

При отправке сообщения критичность события используется как ключ маршрутизации:

```
channel.basic_publish(
    exchange='direct_logs', routing_key=severity, body=message)
```

Программа для приема событий лога будет выглядеть так (receive_logs_direct.py):

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

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

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

    result = channel.queue_declare(queue='', exclusive=True)
    queue_name = result.method.queue

    severities = sys.argv[1:]
    if not severities:
        sys.stderr.write("Usage: %s [info] [warning] [error]\n" % sys.argv[0])
        sys.exit(1)
```

```
    for severity in severities:
        channel.queue_bind(
            exchange='direct_logs', queue=queue_name, routing_key=severity)

    print(' [*] Waiting for logs. To exit press CTRL+C')

    def callback(ch, method, properties, body):
        print(" [x] %r:%r" % (method.routing_key, body))


    channel.basic_consume(
        queue=queue_name, on_message_callback=callback, auto_ack=True)

    channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

В отличие от предыдущей программы, использующей подписку, новая программа в качестве параметров командной строки принимает интересующие нас уровни критичности логируемых событий (info, warning, error) и сохраняет результат в переменной severities:

```
    severities = sys.argv[1:]
    if not severities:
        sys.stderr.write("Usage: %s [info] [warning] [error]\n" % sys.argv[0])
        sys.exit(1)
```

Далее для каждого интересующего нас уровня критичности создаем привязку к очереди через ключ маршрутизаци:

```
    for severity in severities:
        channel.queue_bind(
            exchange='direct_logs', queue=queue_name, routing_key=severity)
```

Callback функция теперь выводит на экран не только само сообщение, но ключ маршрутизации (степень критичности)

```
    def callback(ch, method, properties, body):
        print(" [x] %r:%r" % (method.routing_key, body))
```

В одной консоли запустим программу приема логируемых событий с критичностью warning и error

```
python3 receive_logs_direct.py warning error
```

В другой консоли запустим программу приема логируемых событий с критичностью info, warning и error

```
python3 receive_logs_direct.py info warning error
```

В третьей консоли мы будем публиковать логируемые сообщения. После того как получатели запущены, можно отправлять любое количество сообщений:

```
python3 emit_log_direct.py error "Run. Run. Or it will explode."
python3 emit_log_direct.py info "All OK."
```

Посмотрим, что было доставлено. Один получатель принимает только сообщение об ошибке:

```
 [*] Waiting for logs. To exit press CTRL+C
 [x] 'error':b'Run. Run. Or it will explode.'
```

Второй получатель принимает оба сообщения:

```
 [*] Waiting for logs. To exit press CTRL+C
 [x] 'error':b'Run. Run. Or it will explode.'
 [x] 'info':b'All OK.'
```

Не закрывая получателей, посмотрим, какие существуют точки обмена:

```
sudo rabbitmqctl list_exchanges
```

и видим нашу точку обмена direct_logs, имеющую тип direct:

```
Listing exchanges for vhost / ...
name    type
amq.fanout      fanout
amq.headers     headers
amq.match       headers
direct_logs     direct
amq.rabbitmq.trace      topic
        direct
amq.topic       topic
amq.direct      direct
```

Посмотрим привязки:

```
sudo rabbitmqctl list_bindings
```

и видим, что к точке обмена direct_logs привязаны две очереди со случайными именами:

```
Listing bindings for vhost /...
source_name source_kind    destination_name  destination_kind   routing_key arguments
    exchange    amq.gen-Qu8T3OJpJccW_bHEtYZlXQ  queue  amq.gen-Qu8T3OJpJccW_bHEtYZlXQ []
    exchange    amq.gen-GnIs3dDI9y7zIqFLxVEytw  queue  amq.gen-GnIs3dDI9y7zIqFLxVEytw []
    exchange    hello   queue   hello   []
direct_logs exchange   amq.gen-GnIs3dDI9y7zIqFLxVEytw  queue  error   []
```

```
direct_logs  exchange   amq.gen-Qu8T3OJpJccW_bHEtYZlXQ  queue  error   []
direct_logs  exchange   amq.gen-Qu8T3OJpJccW_bHEtYZlXQ  queue  info    []
direct_logs  exchange   amq.gen-GnIs3dDI9y7zIqFLxVEytw  queue  warning[]
direct_logs  exchange   amq.gen-Qu8T3OJpJccW_bHEtYZlXQ  queue  warning[]
```

Одна очередь использует в качестве ключей привязки (routing_key – binding routing key) info, warning, error, а вторая warning и error.

3.8 Маршрутизация сообщений по маске

Использование точки обмена типа direct дало нам возможность формировать определенные выборки сообщений, однако критерии отбора сообщений недостаточно гибкие. Нам хотелось бы выбирать сообщения не только по их критичности, но и по источнику сообщения. Например, мы хотим отобрать все сообщения логирования с типом error, пришедшие из «cron» и «kern».

Для этого воспользуемся точкой обмена типа topic. Сообщения, отправляемые в точку обмена topic, не могут отсылаться с произвольным ключом маршрутизации (routing_key). Ключ маршрутизации должен представлять собой список слов, разделенный точкой. Слова могут быть любыми, но обычно они ассоциируются с какими-либо свойствами сообщения. Вот примеры правильного routing_key: «stock.usd.nyse», «nyse.vmw», «quick.orange.rabbit». Длина ключа не должна превышать 255 байт.

Ключ привязки (binding key) должен иметь такую же структуру и составляется по такому же правилу. Логика работы topic такая же, как и у direct: сообщения отправляются в очереди, ключ привязки которых совпадает с ключом маршрутизации сообщения. Однако теперь ключ привязки может содержать маску, в которой одно или несколько слов заменены специальными символами * или #. При определении соответствия между ключом маршрутизации и ключом привязки:

− * соответствует одному произвольному слову ключа маршрутизации;

− # соответствует 0 или более слов ключа маршрутизации, разделенных точками.

В примере на рисунке 11 мы отправляем письма о животных. Сообщения содержат ключ маршрутизации, состоящий из трех слов (с двумя точками). Первое слово характеризует скорость, второе – цвет и третье – вид: «speed.colour.species».

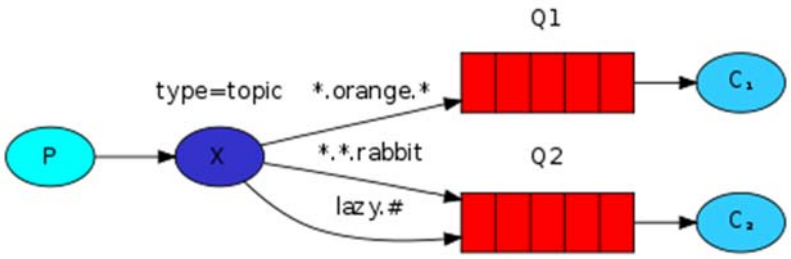

Рисунок 11 – Маршрутизация сообщений по маске

Очередь Q1 будет связана с ключом «*.orange.*», а очередь Q2 – с «*.*.rabbit» и «lazy.#».

Связи могут быть обобщены как:

− очередь Q1 интересуется всеми оранжевыми животными;

− очередь Q2 хочет все знать о кроликах и о ленивых животных.

Сообщение с ключом «quick.orange.rabbit» дойдет до обеих очередей. Сообщение с ключом «lazy.orange.elephant» также придет в обе очереди. Сообщение с типом «quick.orange.fox» дойдет только до очереди Q1, а с типом «lazy.brown.fox» только до Q2. Сообщение с типом «lazy.pink.rabbit» дойдет до очереди Q2 только один раз, хоть и соответствует двум связям. Сообщение с типом «quick.brown.fox» не дойдет ни до одной очереди.

Что будет, если послать сообщение с ключом, не соответствующее ни одному правилу, например «quick.orange.male.rabbit». Такое сообщение никуда не придет и удалится.

А вот сообщение с ключом «lazy.orange.male.rabbit, хотя и состоит из четырех слов, соответствует последней связи и попадет во вторую очередь.

Topic очень мощная точка обмена. Она может работать как другие точки обмена. Если в binding key поместить #, то она поведет себя как fanout. А если не использовать символы * и #, то будет себя вести как direct.

Модифицируем программы так, чтобы они отправляли и принимали сообщения логирования, включающие в себя источник сообщения и его критичность. В качестве ключа маршрутизации будем использовать комбинацию «источник.критичность».

Программа для отправки логируемых событий (emit_log_topic.py) практически идентична предыдущей, только изменены имя точки обмена, ее тип, значение сообщения по умолчанию и имя переменной, содержащей ключ маршрутизации:

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

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

channel.exchange_declare(exchange='topic_logs', exchange_type='topic')

routing_key = sys.argv[1] if len(sys.argv) > 1 else 'anonymous.info'
message = ' '.join(sys.argv[2:]) or 'Hello World!'
channel.basic_publish(
    exchange='topic_logs', routing_key=routing_key, body=message)
print(" [x] Sent %r:%r" % (routing_key, message))
connection.close()
```

Программа для приема событий лога (receive_logs_topic.py) тоже практически идентична предыдущей, только изменены имя точки обмена, ее тип, значение сообщения по умолчанию, имя переменной, содержащей ключ привязки и текст выдаваемого сообщения:

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

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

    channel.exchange_declare(exchange='topic_logs', exchange_type='topic')

    result = channel.queue_declare(queue='', exclusive=True)
    queue_name = result.method.queue

    binding_keys = sys.argv[1:]
    if not binding_keys:
        sys.stderr.write("Usage: %s [binding_key]...\n" % sys.argv[0])
        sys.exit(1)

    for binding_key in binding_keys:
        channel.queue_bind(
            exchange='topic_logs', queue=queue_name, routing_key=binding_key)

    print(' [*] Waiting for logs. To exit press CTRL+C')

    def callback(ch, method, properties, body):
        print(" [x] %r:%r" % (method.routing_key, body))


    channel.basic_consume(
        queue=queue_name, on_message_callback=callback, auto_ack=True)
```

```
    channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('Interrupted')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

В одной консоли запустим программу приема всех логируемых событий

```
python3 receive_logs_topic.py "#"
```

В другой консоли запустим программу приема всех логируемых событий, приходящих от ядра (kern), и критических событий (critical)

```
python3 receive_logs_topic.py "kern.*" "*.critical"
```

В третьей консоли мы будем публиковать логируемые сообщения. После того как получатели запущены, можно отправлять любое количество сообщений:

```
python3 emit_log_topic.py "kern.critical" "A critical kernel error"
python3 emit_log_topic.py "cron.critical" "A critical cron error"
python3 emit_log_topic.py "kern.warning" "A kernel warning"
python3 emit_log_topic.py "cron.warning" "A cron warning"
```

Посмотрим, что было доставлено. Первый получатель принял все сообщения:

```
 [*] Waiting for logs. To exit press CTRL+C
 [x] 'kern.critical':b'A critical kernel error'
 [x] 'cron.critical':b'A critical cron error'
 [x] 'kern.warning':b'A kernel warning'
 [x] 'cron.warning':b'A cron warning'
```

Второй получатель не принял последнее сообщение:

```
 [*] Waiting for logs. To exit press CTRL+C
 [x] 'kern.critical':b'A critical kernel error'
 [x] 'cron.critical':b'A critical cron error'
 [x] 'kern.warning':b'A kernel warning'
```

Не закрывая получателей, посмотрим, какие существуют точки обмена:

```
sudo rabbitmqctl list_exchanges
```

и видим нашу точку обмена topic_logs, имеющую тип topic:

```
Listing exchanges for vhost / ...
name    type
```

```
amq.fanout      fanout
amq.headers     headers
amq.match       headers
direct_logs     direct
topic_logs      topic
amq.rabbitmq.trace      topic
        direct
amq.topic       topic
amq.direct      direct
```

Посмотрим привязки:

```
sudo rabbitmqctl list_bindings
```

и видим, что к точке обмена topic_logs привязаны три очереди со случайными именами:

```
Listing bindings for vhost /...
source_name  source_kind    destination_name    destination_kind routing_key   arguments
    exchange    amq.gen-ywEsvrSYczS7Jmgk4xx5qw queue   amq.gen-ywEsvrSYczS7Jmgk4xx5qw []
    exchange    hello   queue   hello   []
    exchange    amq.gen-U0taEf-lWmeTFTJXYP3EsA queue   amq.gen-U0taEf-lWmeTFTJXYP3EsA []
topic_logs  exchange    amq.gen-U0taEf-lWmeTFTJXYP3EsA queue   #   []
topic_logs  exchange    amq.gen-ywEsvrSYczS7Jmgk4xx5qw queue   *.critical  []
topic_logs  exchange    amq.gen-ywEsvrSYczS7Jmgk4xx5qw queue   kern.*  []
```

Одна очередь использует в качестве ключей привязки (routing_key – binding routing key) «#», вторая «*.critical», а третья «kern.*».

4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1 Для чего предназначена система RabbitMQ?
2 Что такое точка обмена?
3 Для чего используется привязка?
4 Чем отличаются точка привязки типа direct и типа topic?
5 Чем отличаются точка привязки типа direct и типа fanout?

Дик Дмитрий Иванович

Система обмена сообщениями RabbitMQ

Методические указания к выполнению лабораторных работ по дисциплине «Технология построения защищённых распределённых приложений»

для студентов, обучающихся по программе специалитета 10.05.03 – Информационная безопасность автоматизированных систем

Редактор О. Г. Алексеева

Библиотечно-издательский центр КГУ.
640020, г. Курган, ул. Советская, 63, стр.4.
Курганский государственный университет.