Messaging with RabbitMQ

https://www.rabbitmq.com/#getstarted

About RabbitMQ

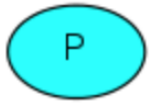
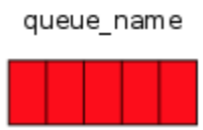
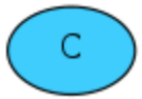
RabbitMQ is an open-source distributed message/streaming broker.

RabbitMQ supports AMQP (Advanced Messaging Queuing Protocol), MQTT (Message Queue Telemetry Transport protocol)

AMQP, MQTT are an open standard application layer protocols.

RabbitMQ accepts, stores, and forwards *messages*: blobs of data

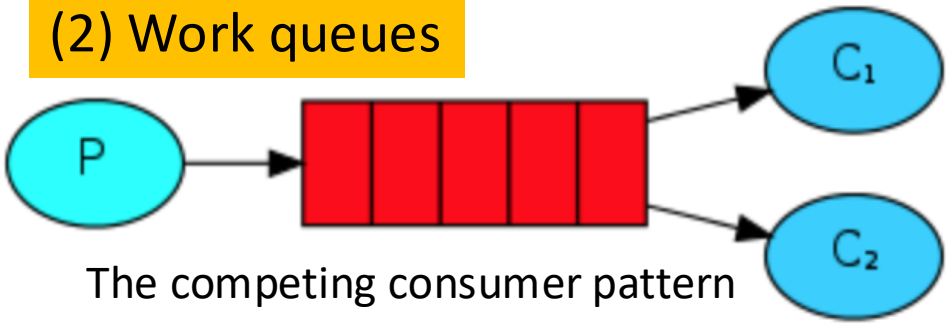
Jargons

Producing A program which sends message is a *producer.*	
Queue : a post box living inside RabbitMQ.	
Consuming A *consumer* is a program that mostly waits to receive messages.	

RabbitMQ is a push model.

(1) Hello world

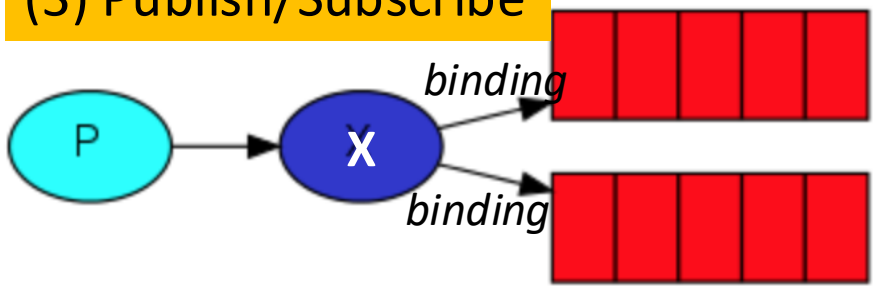
(2) Work queues



The competing consumer pattern

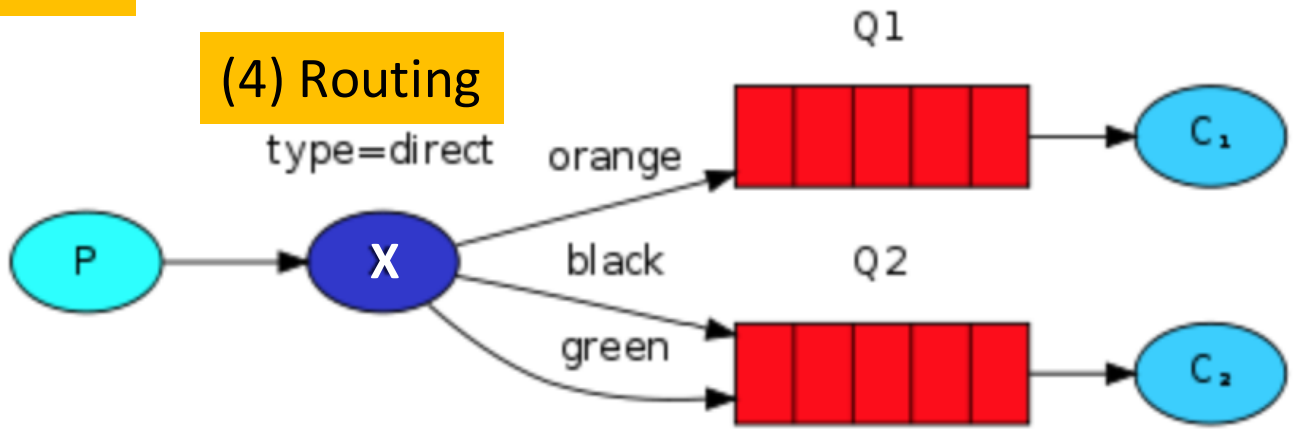
(7) Publisher confirms

(3) Publish/Subscribe



Exchange type: direct, topic, headers, fanout

(4) Routing



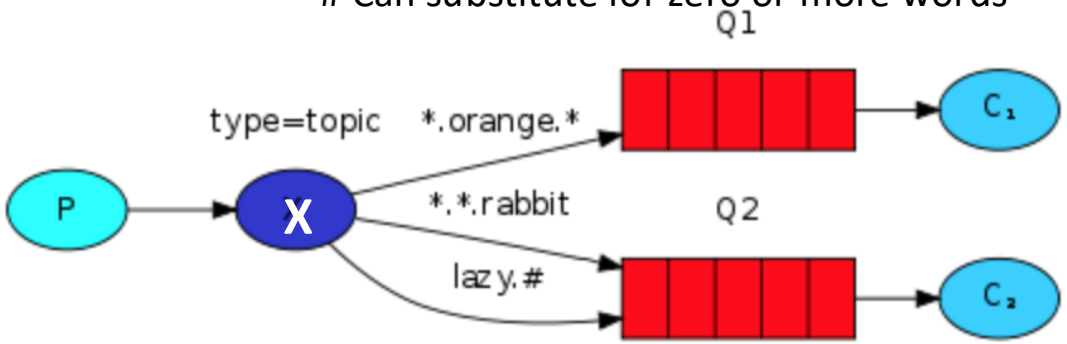
Exchange type: direct, topic, headers, fanout

binding key == routing key

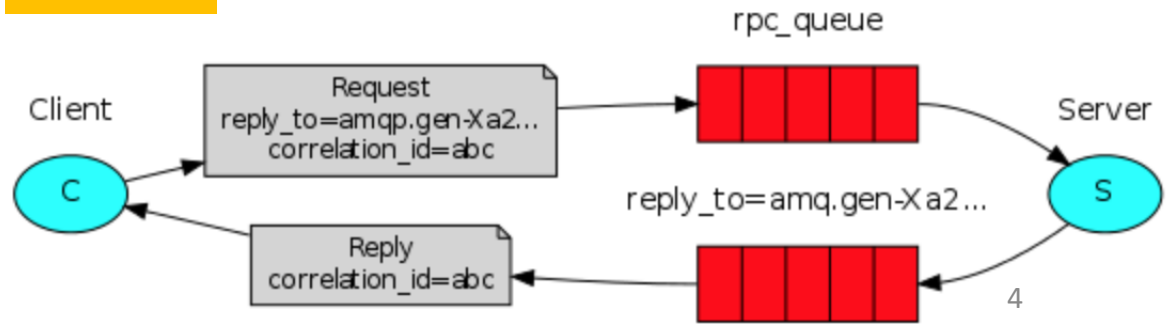
(5) Topics

* Can substitute for exactly one word

\# Can substitute for zero or more words



(6) RPC



Pre-requisite

- RabbitMQ is installed and running on localhost.
- The standard port is 5672.

```
(1)docker run --name my-rabbitmq -p 5672:5672 -p 8080:15672 -d
   rabbitmq:3.13.7
```

http://localhost:8080

```
(2)  docker exec some-rabbit rabbitmqctl status
     docker exec some-rabbit rabbitmqctl
```

https://hub.docker.com/rabbitmq

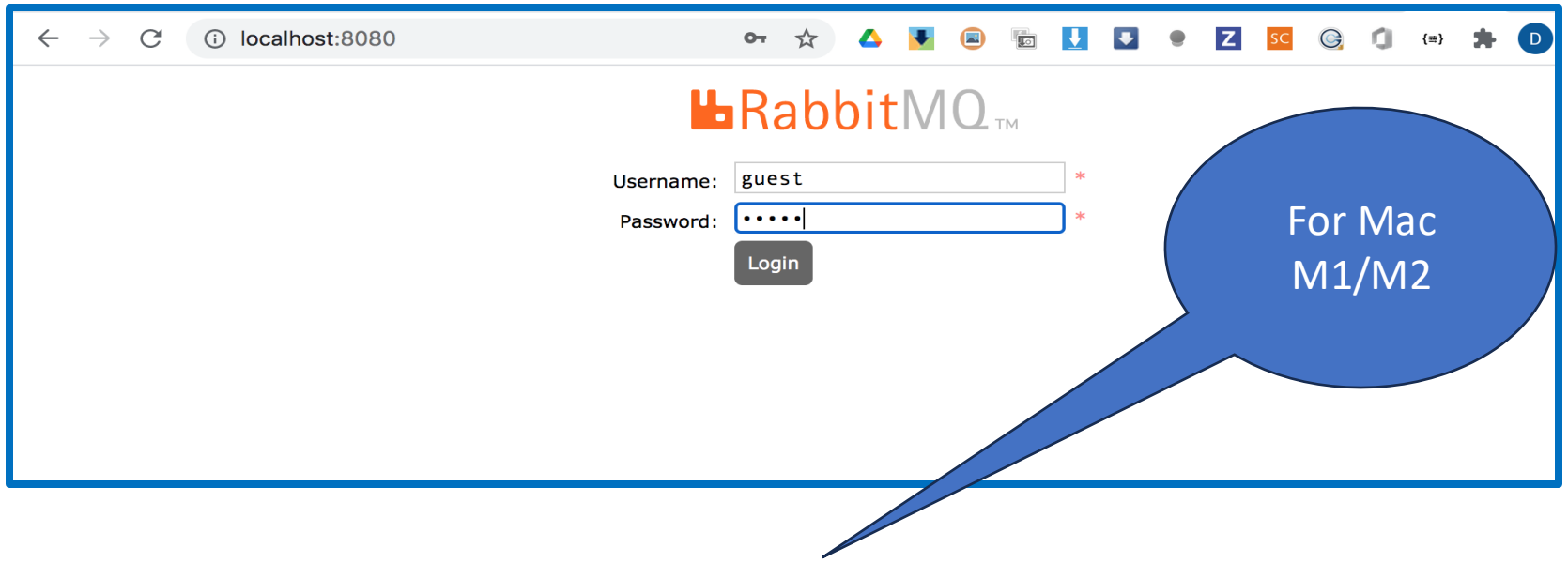
Pre-requisite

For Mac M1/M2

docker-compose.yml

```yaml
services:
  rabbitmq:
    image: 'bitnami/rabbitmq:3.13.7'
    platform: 'linux/arm64'
    container_name: my-rabbitmq
    environment:
      RABBITMQ_ERL_COOKIE: 'securestring'
      RABBITMQ_USERNAME: 'guest'
      RABBITMQ_PASSWORD: 'guest'
      RABBITMQ_VHOST: '/'
      RABBITMQ_PLUGINS: "rabbitmq_management"
    ports:
      - '8080:15672'
      - ‘5672:5672'
```

Pre-requisite




```
docker-compose up

docker exec my-rabbitmq bash -c "echo 'loopback_users.guest = false'
>> /opt/bitnami/rabbitmq/etc/rabbitmq/rabbitmq.conf"

docker-compose restart rabbitmq

docker exec my-rabbitmq bash -c "grep loopback
/opt/bitnami/rabbitmq/etc/rabbitmq/rabbitmq.conf"
loopback_users.guest = false
```

RabbitMQ Management

RabbitMQ™ RabbitMQ 3.9.5 Erlang 24.0.5

Refreshed 2021-09-19 00:38:04 Refresh every 5 seconds

Virtual host All

Cluster **rabbit@my-rabbit**

User **guest** Log out

Overview **Connections** **Channels** **Exchanges** **Queues** **Admin**

Overview

Totals

Queued messages last minute ?

Currently idle

Message rates last minute ?

Currently idle

Global counts ?

Connections: **0** Channels: **0** Exchanges: **7** Queues: **0** Consumers: **0**

Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats	+
rabbit@my-rabbit	37 1048576 available	0 943629 available	393 1048576 available	140 MiB 796 MiB high watermark	32 GiB 48 MiB low watermark	4m 9s	basic disc 2 rss	This node All nodes	

Churn statistics

Ports and contexts

Export definitions

Import definitions

HTTP API **Server Docs** **Tutorials** **Community Support** **Community Slack** **Commercial Support** **Plugins** **GitHub** **Changelog**

Python library as RabbitMQ client

- python -m pip install pika --upgrade

(1)
Hello world

Producer (P)

```python
import sys

#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('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()
```

(1) Hello world


```
python 1_HelloReceive.py
python 1_HelloSent.py
```

(1) Hello world

Consumer (C)

```python
#!/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)
```

(1) Hello world


```
python 1_HelloReceive.py
python 1_HelloSent.py
```

(1)
Hello world

DEMO

(1) Hello world

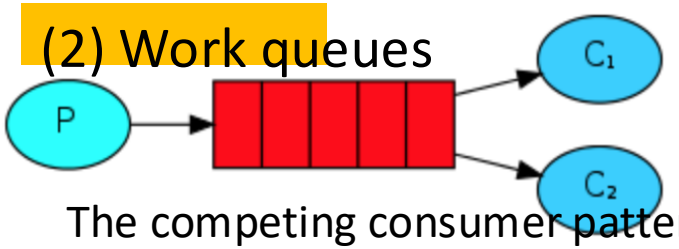

```
python 1_HelloReceive.py
python 1_HelloSent.py
```

(2) Work queue/ Task queue

The assumption behind a work queue is that each task is delivered to exactly one worker.

Round robin dispatching

(2) Work queues



The competing consumer pattern

```
python 2_Worker.py
python 2_Worker.py
…
python 2_TaskQueue.py
```

(2) Work queue/ Task queue (cont.)

Producer (P)

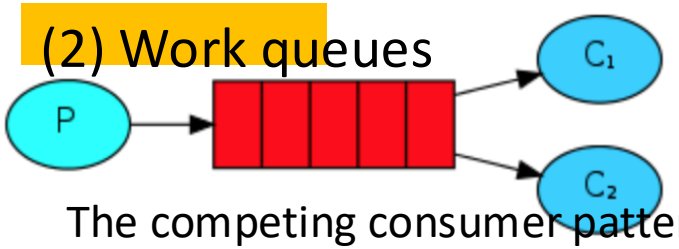
Message durability

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

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

channel.queue_declare(queue='task_queue', durable=True)
                                                  #make queue persistent
message = ' '.join(sys.argv[1:]) or "Hello World!"
channel.basic_publish(
    exchange='',
    routing_key='task_queue',
    body=message,
    properties=pika.BasicProperties(
        delivery_mode=2,  # make message persistent
    ))
print(" [x] Sent %r" % message)
connection.close()
```

To ensure if RabbitMQ quits or crashes, it won't forget queues

pika.spec.PERSISTENT_DELIVERY_MODE

(2) Work queues



The competing consumer pattern

```
python 2_Worker.py
python 2_Worker.py
…
python 2_TaskQueue.py
```

(2) Work queue/ Task queue (cont.)

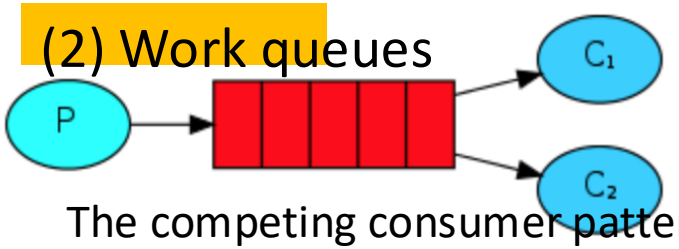
Consumer (C)

Message durability

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

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

channel.queue_declare(queue='task_queue', durable=True)
print(' [*] Waiting for messages. To exit press CTRL+C')


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)

channel.start_consuming()
```

To ensure if RabbitMQ quits or crashes, it won't forget queues

(2) Work queues



The competing consumer pattern

```
python 2_Worker.py
python 2_Worker.py
…
python 2_TaskQueue.py
```

(2) Work queue/ Task queue (cont.)

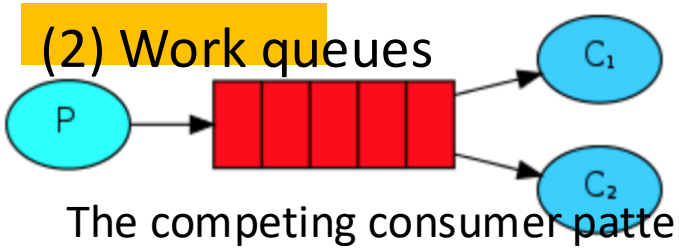
Consumer (C)

Manage acknowledge

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

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

channel.queue_declare(queue='task_queue', durable=True)
print(' [*] Waiting for messages. To exit press CTRL+C')


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)

channel.start_consuming()
```

To ensure that worker will send ack when completing the task.

(2) Work queues

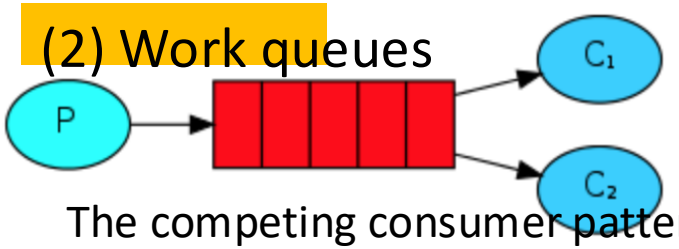


The competing consumer pattern

```
python 2_Worker.py
python 2_Worker.py
…
python 2_TaskQueue.py
```

(2) Work queue/ Task queue (cont.)

(2) Work queues



The competing consumer pattern

```
python 2_Worker.py
python 2_Worker.py
…
python 2_TaskQueue.py
```

Consumer (C)

What the difference between round robin and fair dispatch?

Fair dispatch

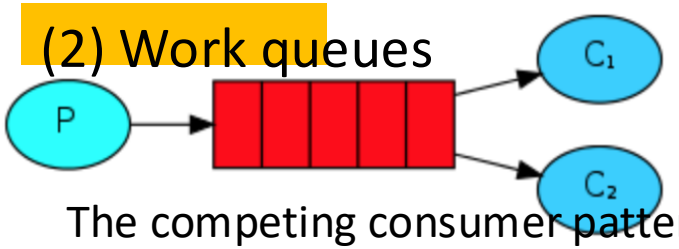
```python
#!/usr/bin/env python
import pika
import time

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

channel.queue_declare(queue='task_queue', durable=True)
print(' [*] Waiting for messages. To exit press CTRL+C')


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)

channel.start_consuming()

```

Tell the RabbitMQ not to dispatch a new task if the worker is busy.

(2) Work queue/ Task queue (cont.)

DEMO

(2) Work queues



The competing consumer pattern

```
python 2_Worker.py
python 2_Worker.py
…
python 2_TaskQueue.py
```

(3) Publish/ Subscribe (fanout)

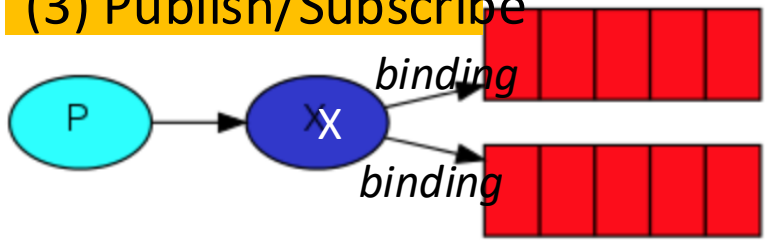
RabbitMQ will deliver a message to multiple consumers.

```
docker exec some-rabbit rabbitmqctl list_exchanges
docker exec some-rabbit rabbitmqctl list_bindings
```

Producer (P)

```python
#!/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()
```

(3) Publish/Subscribe



Exchange type: direct, topic, headers, fanout

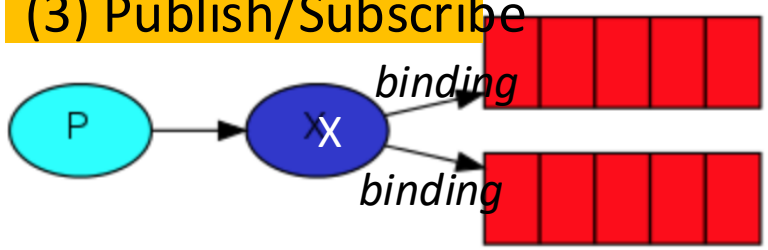
```
python 3_Publish_broadcast_log.py
python 3_Subscribe_log.py
```

The use of exchange introduces the full messaging model in Rabbit.

Core idea is that a producer never send any messages directly to a queue..

(3) Publish/ Subscribe (fanout) (cont.)

(3) Publish/Subscribe



Exchange type: direct, topic, headers, fanout

```
python 3_Publish_ broadcast_log
python 3_Subscribe_log.py
```

Consumer (C)

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

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()
```

We want to hear all messages and only the current flowing messages not the older ones.

Let the server choose the random queue name and this queue will be deleted when the consumer's connection is closed.

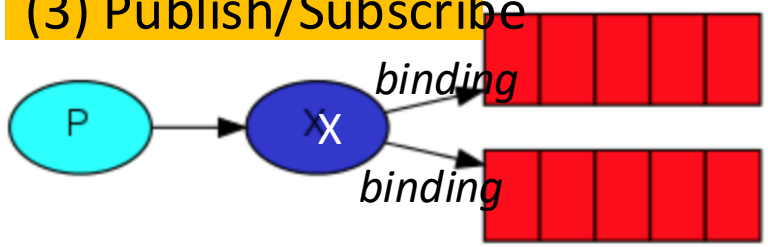
Bind exchange with queue

(3) Publish/ Subscribe (fanout) (cont.)

```
python 3_Subscribe_log.py
python 3_Subscribe_log.py
python 3_Publish_broadcast_log.py
```

DEMO

(3) Publish/Subscribe

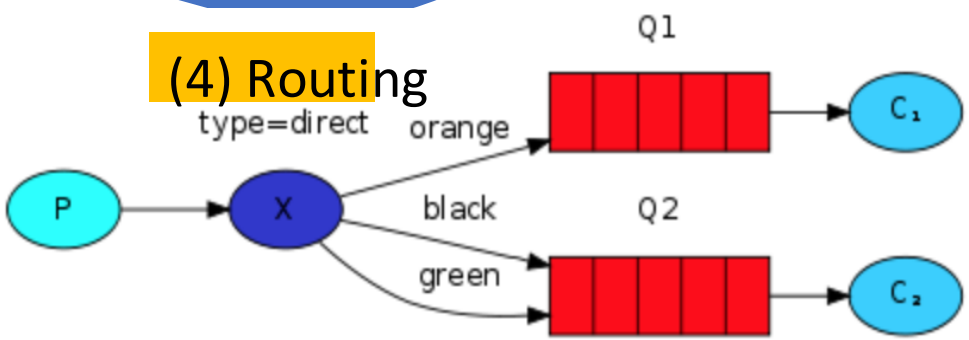


Exchange type: direct, topic, headers, fanout

```
python 3_Publish_broadcast_log.py
python 3_Subscribe_log.py
```

(4) Routing

(4) Routing



Exchange type: direct, topic, headers, fanout

binding key == routing key

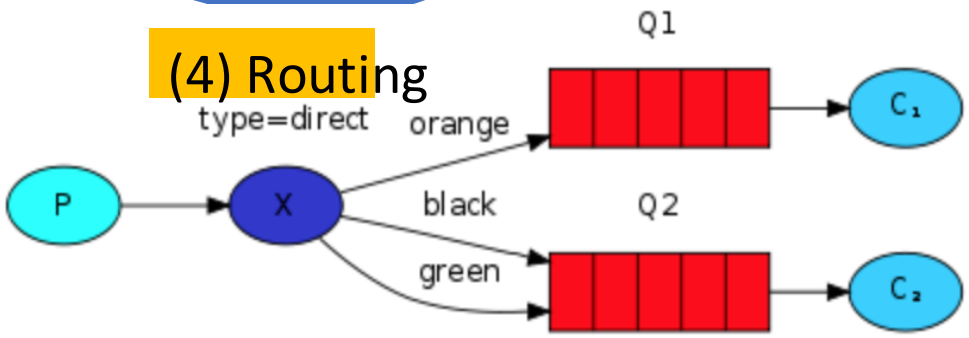
```
python 4_Publish_direct_log.py
python 4_Subscrib_direct_log.py
```

Producer (P)

```python
#!/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()
```

(4) Routing

(4) Routing



Exchange type: direct, topic, headers, fano

binding key == routing key

```
python 4_Publish_direct_log.py
python 4_Subscrib_direct_log.py
```

Consumer (C)

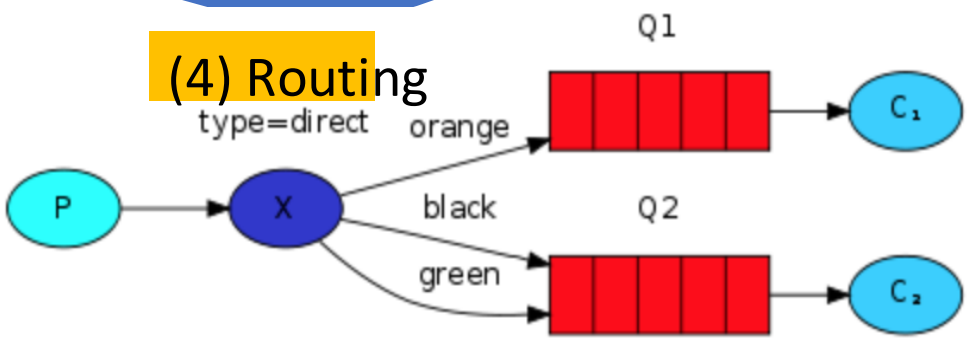
```python
#!/usr/bin/env python
import pika
import sys

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

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

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()
```

(4) Routing

```
python 4_Subscribe_direct_log.py warning error > logs_from_rabbit.log
python 4_Subscribe_direct_log.py info warning error
python 4_Publish_direct_log.py error "GPU card 0 is down!!"
python 4_Publish_direct_log.py info "Add new GPU card!!
```

DEMO

(4) Routing



Exchange type: direct, topic, headers, fanc

binding key == routing key

```
python 4_Publish_direct_log.py
python 4_Subscrib_direct_log.py
```

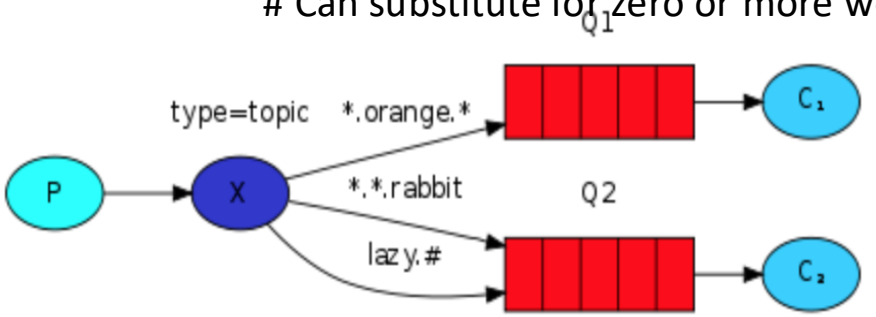
(5) Topic

```python
#!/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) > 2 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()
```

(5) Topics

* Can substitute for exactly one word

\# Can substitute for zero or more words



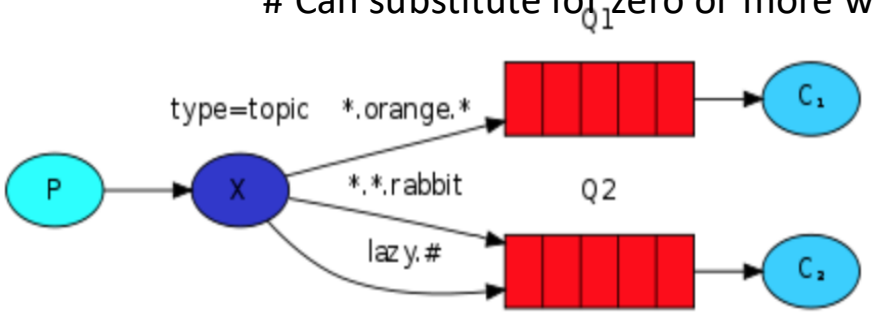

```
python 5_Publish_topic_log.py
python 5_Subscribe_topic_log.py
```

(5) Topic

(5) Topics

* Can substitute for exactly one word

\# Can substitute for zero or more words




```
python 5_Publish_topic_log.py
python 5_Subscribe_topic_log.py
```

```python
#!/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')

result = channel.queue_declare('', 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()
```

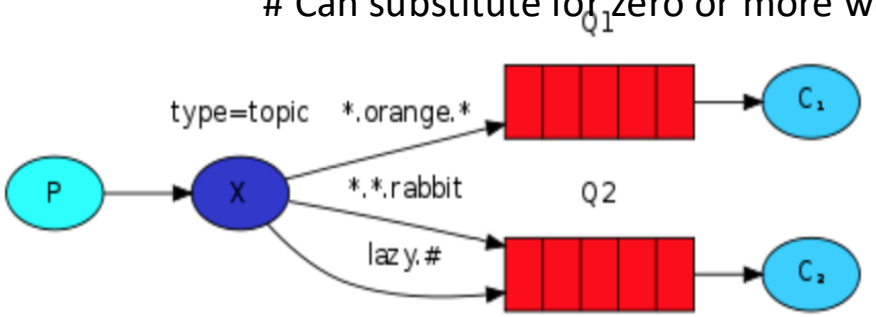
(5) Topic

```
Terminal 1: python 5_Subscribe_topic_log.py "#"
Terminal 2: python 5_Subscribe_topic_log.py "kern.*"
Terminal 3: python 5_Subscribe_topic_log.py "*.critical"
Terminal 4: python 5_Subscribe_topic_log.py "kern.*" "*.critical"
Terminal 5: python 5_Publish_topic_log.py "kern.critical" "kernel error"
```

DEMO

(5) Topics

* Can substitute for exactly one word
\# Can substitute for zero or more words




```
python 5_Publish_topic_log.py
python 5_Subscribe_topic_log.py
```

RabbitMQ & Apache Kafka

RabbitMQ	Kafka
Direct messaging: users can set sophisticated rules for message delivery.	Kafka is ideal for handling large amounts of homogeneous messages, such as logs or metric, and it is the right choice for instances with high throughput.
RabbitMQ will be usually used with Cassandra for message playback.	Replay messages
Multiprotocol supports: AMQP, STOMP, MQTT, Web sockets and others.	Big data consideration with e.g., Elastic search, Hadoop
Flexibility: varied point-to-point, request/reply, publish/subscribe	Scaling capability; topics can be split into partitions
Communication: supports both async and sync	Batches: Kafka works best when messages are batched.
Security	Security
Mature platform	Mature platform
Slower than Kafka	Don’t come with user-friendly GUI but can be monitored via Kibana

https://www.upsolver.com/blog/kafka-versus-rabbitmq-architecture-performance-use-case
https://dattell.com/data-architecture-blog/kafka-vs-rabbitmq-how-to-choose-an-open-source-message-broker/

Summary of differences: Kafka vs. RabbitMQ

	RabbitMQ	Kafka
Architecture	RabbitMQ's architecture is designed for complex message routing. It uses the push model. Producers send messages to consumers with different rules.	Kafka uses partition-based design for real-time, high-throughput stream processing. It uses the pull model. Producers publish messages to topics and partitions that consumers subscribe to.
Message handling	RabbitMQ brokers monitor message consumption. It deletes messages after they're consumed. It supports message priorities.	Consumers keep track of message retrieval with an offset tracker. Kafka retains messages according to the retention policy. There's no message priority.
Performance	RabbitMQ has low latency. It sends thousands of messages per second.	Kafka has real-time transmission of up to millions of messages per second.
Programming language and protocol	RabbitMQ supports a broad range of languages and legacy protocols.	Kafka has limited choices of programming languages. It uses binary protocol over TCP for data transmission.

https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-kafka/

RabbitMQ & Apache Kafka

RabbitMQ: Used for high throughput and reliable background jobs, integration and intercommunication between and within applications, perform complex routing to consumers, integrate multiple applications and services with non-trivial routing logic.

Kafka: Best used for basic streaming without complex routing with maximum throughput, ideal for event-sourcing, stream processing, multi-stage pipelines, routinely audited systems, real-time processing and analyzing data

https://www.simplilearn.com/kafka-vs-rabbitmq-article#what_is_kafka_used_for

Assignment based on RabbitMQ

Preparation

- Download starter code from

https://drive.google.com/drive/folders/1wXvFEEu431o1WMT1_8031MNadlwVyO2L?usp=sharing

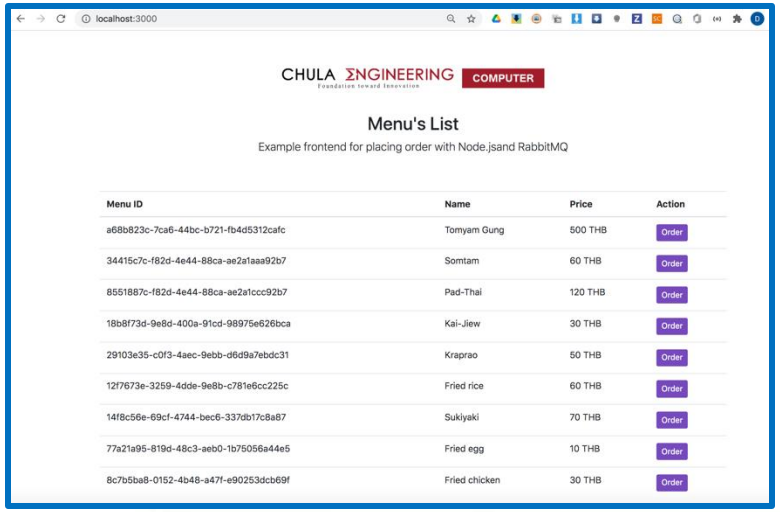
This code was extended from gRPC activity last week with example of simple queue.

```
unzip the restaurant_w_order.zip
npm install amqplib
```

- Start rabbitmq docker as follow

```
docker run -d --name my-rabbitmq -p 5672:5672 -p 8080:15672 rabbitmq:3.13.7
```

Implement the restaurant_w_order using **topic** for desserts, Thai dishes, Italian dishes, drinks as you can imagine.



Work Queue

RabbitMQ

Kitchen 1 — server/node kitchen.js

Kitchen 2 — server/node kitchen.js

Kitchen 3 — server/node kitchen.js

...

server/npm start

JavaScript web app

web client

Browser

HTTP

REST API server

GRPC client

client/
node index.js

Protocol Buffers

GRPC server

GRPC server

Backend

REST

index.js

Introduce the /placeorder as the producer of orders to kitchens

```
var amqp = require('amqplib/callback_api');

app.post("/placeorder", (req, res) => {
    //const updateMenuItem = {
    var orderItem = {
        id: req.body.id,
        name: req.body.name,
        quantity: req.body.quantity,
    };

    // Send the order msg to RabbitMQ
    amqp.connect('amqp://localhost', function(error0, connection) {
        if (error0) {
            throw error0;
        }
        connection.createChannel(function(error1, channel) {
            if (error1) {
                throw error1;
            }
            var queue = 'order_queue';
            //var msg = process.argv.slice(2).join(' ') || "Hello World!";

            channel.assertQueue(queue, {
                durable: true
            });
            channel.sendToQueue(queue, Buffer.from(JSON.stringify(orderItem)), {
                persistent: true
            });
                console.log(" [x] Sent '%s'", orderItem);
            });
        });
    });
```

kitchen.js

The consumer of the orders

```javascript
#!/usr/bin/env node

var amqp = require('amqplib/callback_api');

amqp.connect('amqp://localhost', function(error0, connection) {
  if (error0) {
    throw error0;
  }
  connection.createChannel(function(error1, channel) {
    if (error1) {
      throw error1;
    }
    var queue = 'order_queue';

    channel.assertQueue(queue, {
      durable: true
    });
    channel.prefetch(1);
    console.log(" [*] Waiting for messages in %s. To exit press CTRL+C", queue);
    channel.consume(queue, function(msg) {
        var secs = msg.content.toString().split('.').length - 1;
        console.log(" [x] Received");
        console.log(JSON.parse(msg.content));

        setTimeout(function() {
        console.log(" [x] Done");
        channel.ack(msg);
        }, secs * 1000);
    }, {
    noAck: false
    });
  });
});
```

Things to be delivered

A video clip demonstrating

(1) The communication between the producer (index.js) and the consumers (kitchen.js) with various order of foods

(2) The queue monitoring via RabbitMQ-Management tool.

RabbitMQ as a Service

CloudAMQP

Pricing Documentation Support Blog About us

Login Sign Up

RabbitMQ as a Service

Managing the largest fleet of RabbitMQ clusters in the world

Get a managed RabbitMQ server today

SHARED INSTANCES

For development or small hobby projects. Not recommended for production due to variable performance.

Tough Tiger - For Hobby Apps

- Max 1000 queues
- Max 100 000 queued messages

10 M	100
Max MSGs/month	Max connections

$ 19

PER MONTH

Get Now

Little Lemur - For Development

- Max 100 queues
- Max 10 000 queued messages
- Max idle queue time 28 days

1 M	20
Max MSGs/month	Max connections

FREE

Get Now

สร้าง account ที่ cloudAMQP

https://www.cloudamqp.com/plans.html

CloudAMQP

List all instances

SWARCH

Instances

+ Create New Instance

Name	Plan	Datacenter	Actions
You don't have any instances yet, do you want to create one?			

CloudAMQP

List all instances

SWARCH

Create new instance

No credit card Please add a credit card if you want to subscribe to a paid plan

Missing billing information Please fill in all required information if you want to subscribe to a paid plan

Plan | Region | Configure (Dedicated plans only) | Confirm

Select a plan and name - Step 1 of 4

Name: For_SW_ARCH_Practice

Plan: Little Lemur (Free)

Tags:

Tags are used to separate your instances between projects. This is primarily used in the project listing view for easier navigation and access control.

Tags allow admins to manage team members access to different groups of instances.

Plan

Little Lemur

See the plan page to learn about the different plans.

Cancel | Select Region

CloudAMQP

List all instances

SWARCH

Create new instance

No credit card Please add a credit card if you want to subscribe to a paid plan

Missing billing information Please fill in all required information if you want to subscribe to a paid plan

Plan — Region — Configure (Dedicated plans only) — Confirm

Select a region and data center - Step 2 of 4

Data center: US-West-2 (Oregon)

aws

Free instance เลือกที่อื่น เช่น Singapore ไม่ได้

Plan

Little Lemur

See the plan page to learn about the different plans.

« Back

Cancel Review

Plan Region Configure (Dedicated plans only) Confirm

Confirm new instance - Step 4 of 4

Plan

Little Lemur

Total: $0 / month

Name:	For_SW_ARCH_Practice
Platform:	Amazon Web Services
Region:	US-West-2 (Oregon)

ตรวจดูความเรียบร้อย

« Back

Cancel

Create instance

เสร็จเรียบร้อย

CloudAMQP

List all instances

SWARCH

Instances

+ Create New Instance

Name	Host	Plan	Datacenter	Actions
For_SW_ARCH_Practice	cattle	Little Lemur	Amazon Web Services US-West-2 (Oregon)	Edit RabbitMQ Manager

Showing 1 to 1 of 1 entries