RabbitMQ : A MOM (Message Oriented Middleware)

Yvon Kermarrec

With figures from RabbitMQ tutorial and from “TabbitMQ in action”

Bibliography

- https://www.amqp.org/
- https://www.rabbitmq.com/
- « RabbitMQ in action » Alvaro Videla and Jason J.W. Williams
- « learning RabbitMQ » Martin Toshev
- « RabbitMQ essentials » David Dossot

Agenda

- AMQP
- Understanding messaging
- Using RabbitMQ illustrated with examples
- Administration of the broker
- Synthesis

Why AMQP ?

- Business, applications, services and devices need to interact and communicate information
- IBM technologies controled over 80% of the market with the WebSphere and MQ
- Hughe cost and danger of a monopoly…

AMQP

- The Advanced Message Queuing Protocol (AMQP) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security (wikipedia)

AMQP

- AMQP provides a platform-agnostic method for ensuring information is safely transported between applications, among organizations, within mobile infrastructures, and across the Cloud.
- AMQP is used in areas as varied as financial front office trading, ocean observation, transportation, smart grid, computer-generated animation, and online gaming. Many operating systems include AMQP implementations, and many application frameworks are AMQP-aware

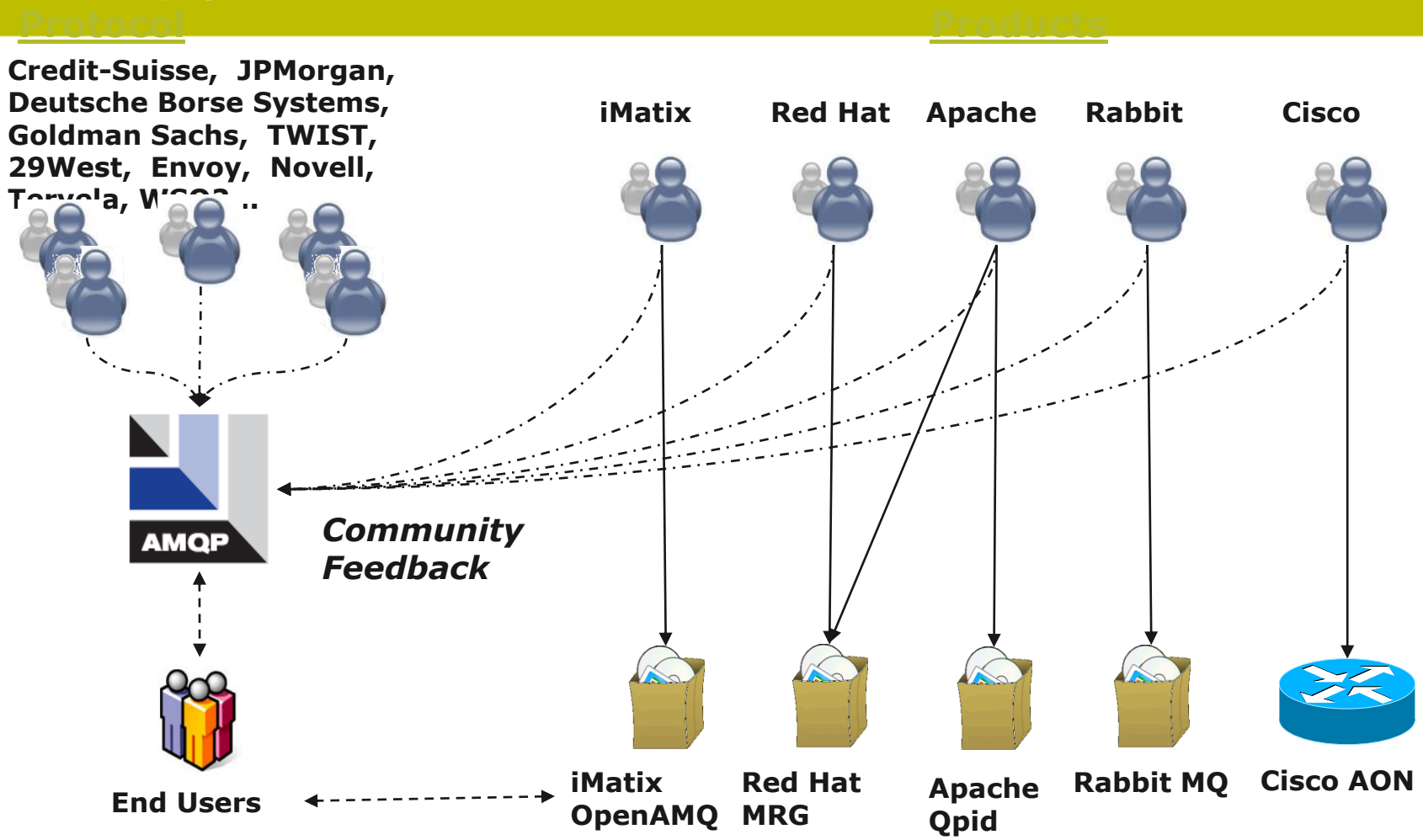
AMQP user group

- **Set up by JPMorgan in 2006**
  - Goal to make Message Oriented Middleware pervasive
  - Make it practical, useful, interoperable
- **The working group grew to 23 companies including Bank of America, Barclays, Cisco Systems, Credit Suisse, Deutsche Börse, Goldman Sachs, HCL Technologies Ltd, Progress Software, IIT Software, INETCO Systems Limited, Informatica JPMorgan Chase, Microsoft Corporation, my-Channels, Novell, Red Hat, Software AG, Solace Systems, StormMQ, Tervela Inc., TWIST Process Innovations ltd, VMware**

AMQP user group

- **MOM needs to be *everywhere* to be useful**
  - dominant solutions are proprietary
    - too expensive for everyday use (Cloud-scale)
    - they don't interoperate
  - has resulted in lots of ad-hoc home-brew
- **Middleware Hell**
  - 100's of applications - 10,000's of links
  - every connection different - massive waste of effort
- **The Internet's missing standard**

AMQP Working Group – Strong Governance



AMQP Working Group controls the standard

Diverse products implement the standard

TELECOM Bretagne

Agreed User Requirements

- **UBIQUITOUS AND PERVASIVE**
  - Open internet protocol standard
  - Binary WIRE protocol so that it can be ubiquitous, fast, embedded
  - Unambiguous core functionality for business message routing and delivery within Internet infrastructure
  - Scalable, so that it can be a basis for high performance fault-tolerant lossless messaging infrastructure, i.e without requiring other messaging technology
  - Fits into existing enterprise messaging applications environments in a practical way

Agreed User Requirements

- **UBIQUITOUS AND PERVASIVE**
- **SAFETY**
  - Infrastructure for a secure and trusted global transaction network
    - Consisting of business messages that are tamper proof
    - Supporting message durability independent of receivers being connected
  - Transport business transactions of any financial value
  - Sender and Receiver are mutually agreed upon counter parties
    - No possibility for injection of Spam

Agreed User Requirements

- **UBIQUITOUS AND PERVASIVE**
- **SAFETY**
- **FIDELITY**
  - Well-stated message queuing and delivery semantics covering
    - at-most-once - at-least-once and once-and-only-once (e.g. 'reliable', 'assured', 'guaranteed')
  - Well-stated message ordering semantics describing what a sender can expect
    - a receiver to observe
    - a queue manager to observe
  - Well-stated reliable failure semantics
    - so exceptions can be managed

Agreed User Requirements

- **UBIQUITOUS AND PERVASIVE**
- **SAFETY**
- **FIDELITY**
- **UNIFIED**
  - AMQP aspires to be the sole business messaging tool for organizations
  - Global addressing standardizing end-to-end delivery across any network scope
  - Any AMQP client can initiate communication with, and then communicate with, any AMQP broker over TCP/IP
  - Optionally, extendable to alternate transports via negotiation

Agreed User Requirements

- **UBIQUITOUS AND PERVASIVE**
- **SAFETY**
- **FIDELITY**
- **UNIFIED**
  - Provide a core set of messaging patterns via a single manageable protocol:
    - asynchronous directed messaging
    - request/reply, publish/subscribe
    - store-and-forward
  - Provide for Hub-and-Spoke messaging topology within and across business boundaries
  - Provide for Hub-to-Hub message relay across business boundaries through enactment of explicit agreements between broker authorities
  -

Agreed User Requirements

- **UBIQUITOUS AND PERVASIVE**
- **SAFETY**
- **FIDELITY**
- **UNIFIED**
- **INTEROPERABILITY**

  - Multiple stable and interoperating broker implementations
    - Each with a completely independent provenance (min. 2 to move to Final)
    - Each broker implementation is conformant with the specification, for all mandatory functionality, including fidelity semantics

  - Layered architecture, so features & network transports can be independently extended by separated communities of use

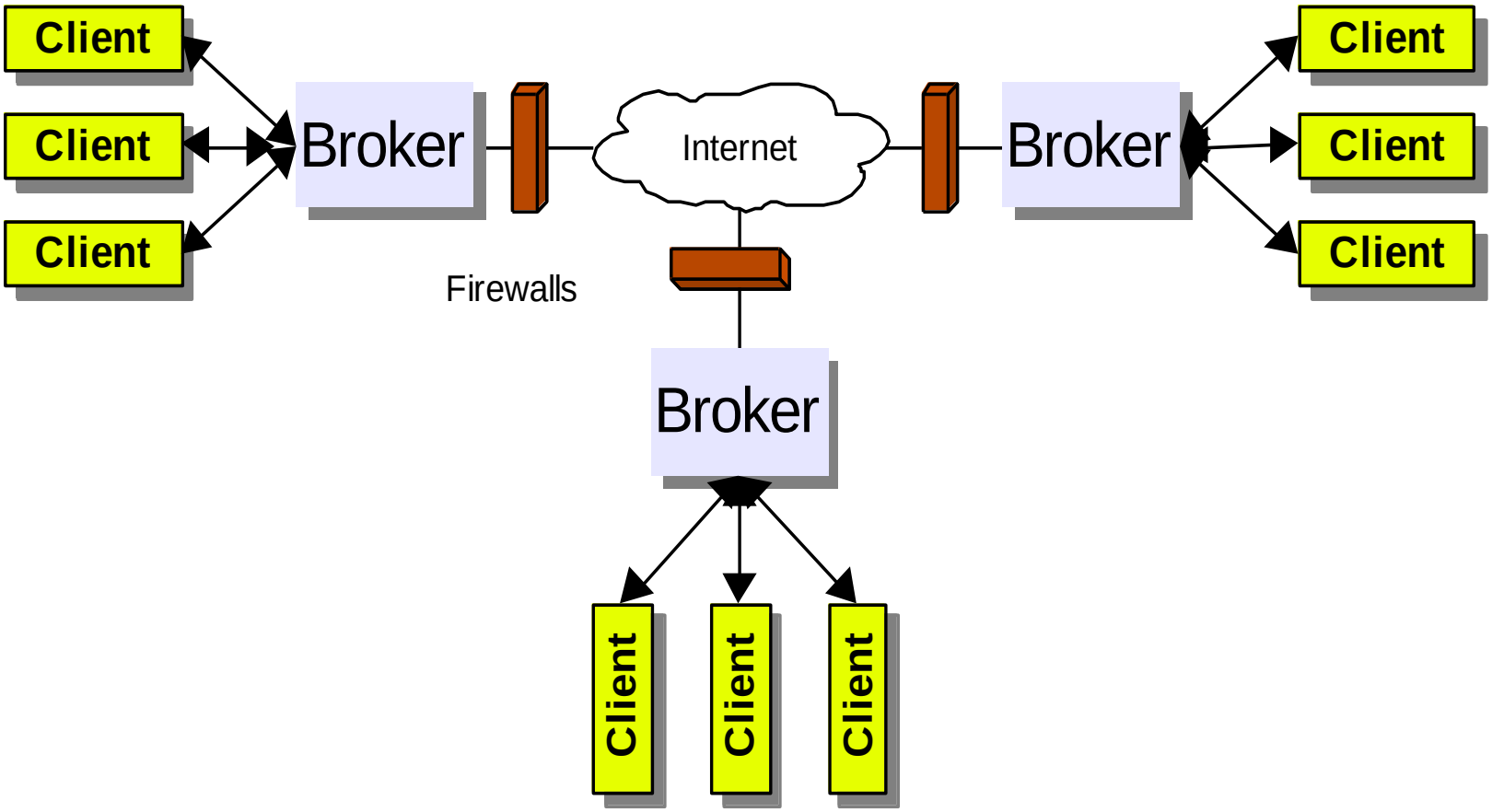
Agreed User Requirements

- **UBIQUITOUS AND PERVASIVE**
- **SAFETY**
- **FIDELITY**
- **UNIFIED**
- **INTEROPERABILITY**
- **MANAGEABLE**

Banking Security Requirements

- SSL support
- Service Context (incl. Security Context):
- Support for carrying Security Tokens:
- Unique Security Token per Message:
- Hash and sign of Message (including Security Context)
  - Assure authenticity of the contents in addition to encryption (content verified by final-destination).
  - Full-path privacy for business transactions that might pass through a number of hubs enroute to the final destination, where you would not want to have the exposed content of the message sitting in some queue and disk along the way.
- Chains of trust within trust realms

Inter-Network Connectivity



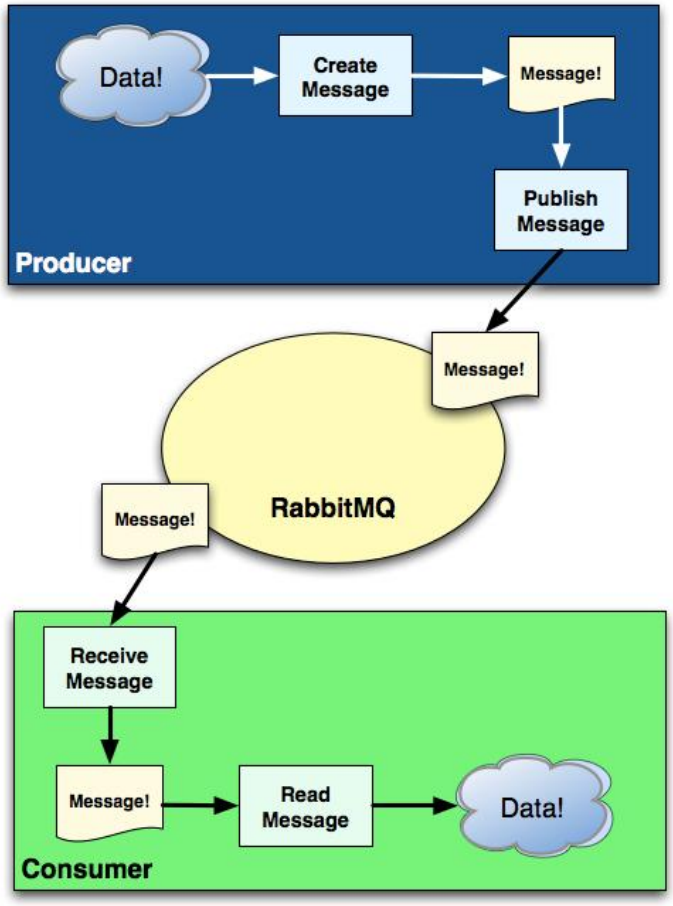
Various implementations

- RabbitMQ: http://www.rabbitmq.com/
- ZeroMQ: http://zeromq.org/
- ActiveMQ: http://activemq.apache.org/
- ….

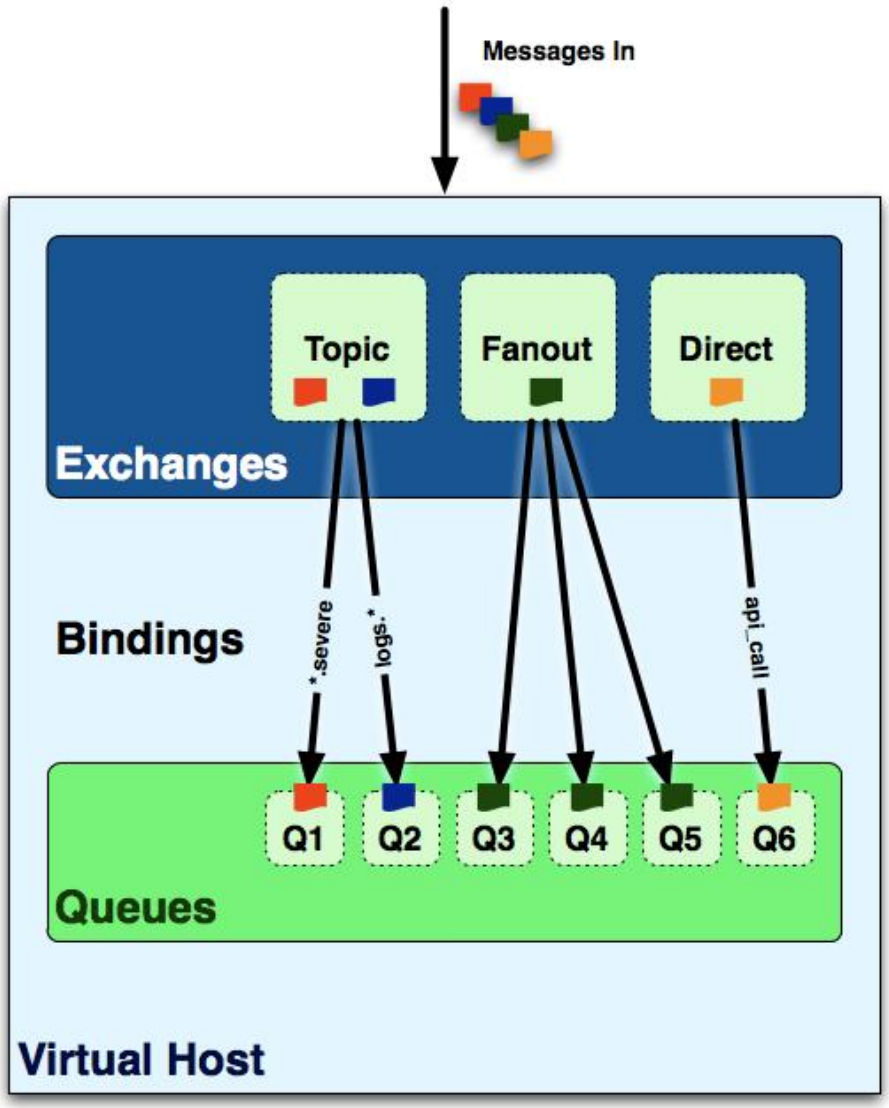
Agenda

- AMQP
- Understanding messaging
- Using RabbitMQ illustrated with examples
- Administration of the broker
- Synthesis

RabbitMQ in context



RabbitMQ in context



Queues

- Queues are like named mailboxes.
- A queue can be declared several times with no impact (to avoid who should be launched first)
- They are where messages end up and wait to be consumed
- Consumers can receive messages from a specific queue
- If there are one or more consumers subscribed to a queue, messages are sent immediately to the subscribed consumers.

Queues

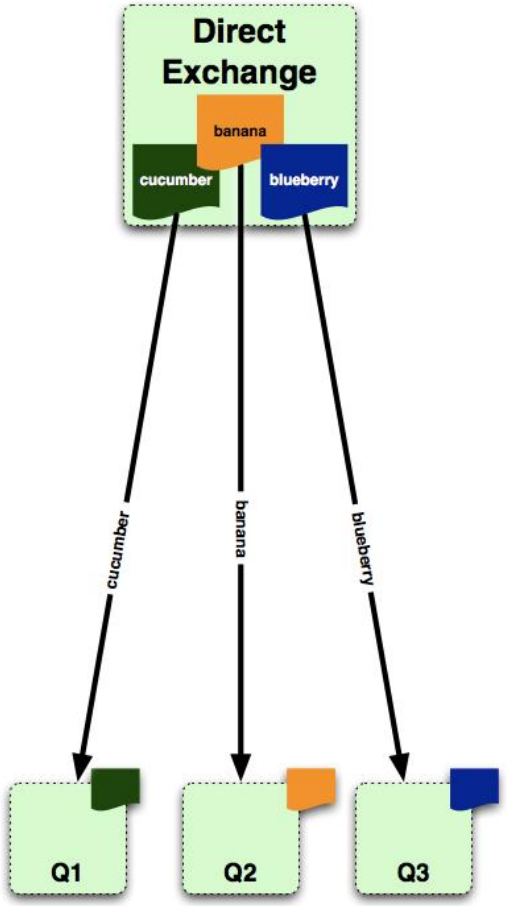
- If a message arrives at a queue with no subscribed consumers, the message waits in the queue.
- When a queue has multiple consumers, messages received by the queue are served in a round-robin fashion to the consumers
- Each message is sent to only one consumer subscribed to the queue.
- Every message that is received by a consumer is required to be acknowledged (implicitly with ‘auto_ack’ or explictly with an ‘Ack’) and will then be removed from the queue
- Acks are used to control the flow of messages

Exchanges and bindings

- When we want to deliver a message to a queue, we do it by sending it to an exchange.
- RabbitMQ will decide to which queue it should deliver the message (by applying rules or routing rules)
- The 4 different routing approaches: direct, fanout, topic and header

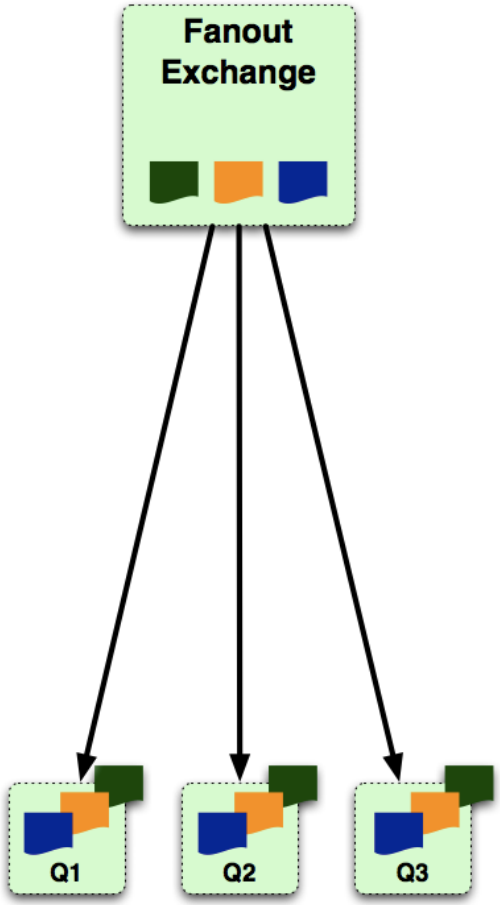
Bindings : direct exchange – use the queue name as the routing key

When a queue is declared it will be automatically bound to that exchange using the queue name as routing key



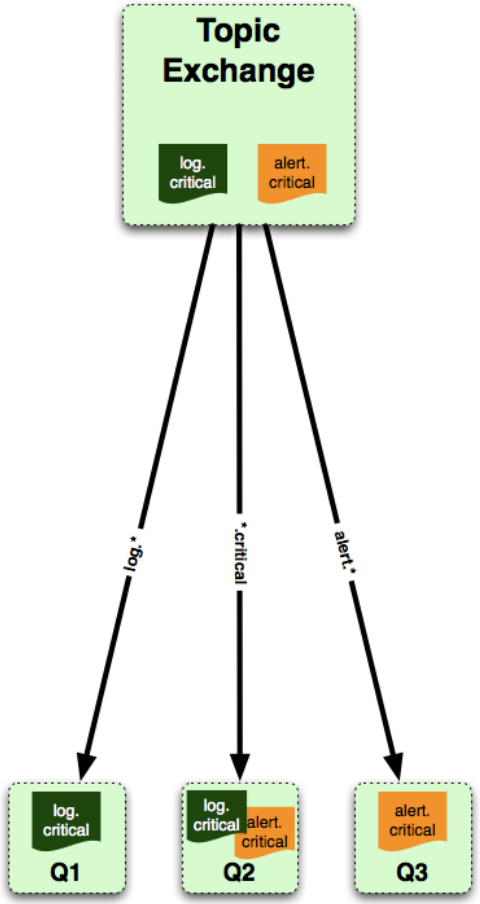
Bindings : fanout or multicast

when we send a message to a fanout exchange, it will be delivered to all the queues attached to this exchange



Binding with topic

A message can be routed based on the key and the rule is expressed on specific names or "*"



Message durability and persistence

- **Message are routed to queues**
- **The exchanges and queues can be set to ‘durable’**
  - By default to non durable – so when the systems reboots it forgets / erases all of them
  - If set to true, the exchanges and queues will be restored
- **A message can be set to persistent or not (delivery mode flag set to true or false)**
- **A message will survive a reboot, if its "delivery mode" option is set to persistent(value of 2), be published into a durable exchange and arrive in a durable queue**

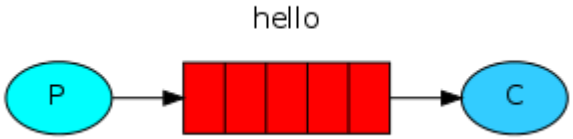
Summary of messaging

- The key components in the AMQP architecture are exchanges, queues and bindings.
- We bind queues to exchanges based on routing keys.
- Messages are sent to exchanges.
- There are three major exchange types, direct, fanout and topic.
- Based on the message routing key and the exchange type, the broker will decide to which queue it has to deliver to message.
- Message can be set to persistent ; queues and exchanges to durable

Agenda

- AMQP
- Understanding messaging
- Using RabbitMQ illustrated with examples
- Administration of the broker
- Synthesis

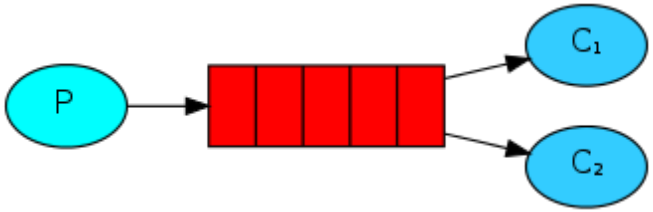
An initial example



Producer side:

```
#!/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()
```

A second example

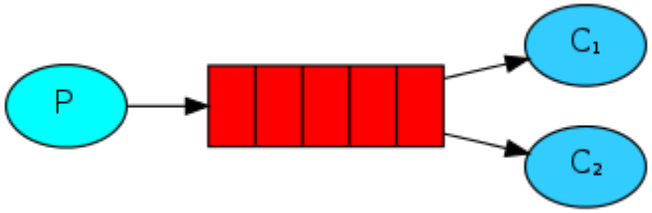


Consumer side:

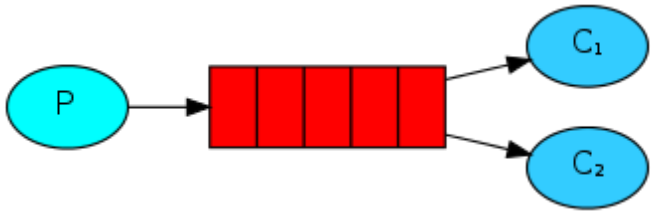
```
 #!/usr/bin/env python
import pika

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

channel.queue_declare(queue='hello')

print ' [*] Waiting for messages. To exit press CTRL+C'

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

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

channel.start_consuming()
```

A second example

- Classic master and slaves paradigm: a master split some workload and the ‘willing’ slaves get their load



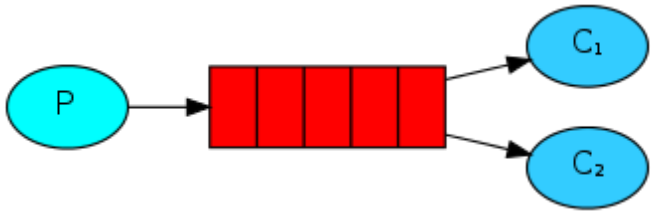
A second example



Worker side:

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

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,)
    time.sleep( body.count('.') )
    print " [x] Done"
    ch.basic_ack(delivery_tag = method.delivery_tag)

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

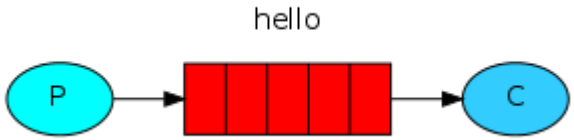
A second example



Producer side:

```
#!/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)

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()
```

An initial example



Consumer side:

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

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

channel.queue_declare(queue='hello')

print ' [*] Waiting for messages. To exit press CTRL+C'

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

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

channel.start_consuming()
```

An example with an exchange: a logger

- A producer can only send messages to an *exchange*.
- An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues.
- The exchange must know exactly what to do with a message it receives
- There are a few exchange types available: direct, topic, headers and fanout (or multicast)

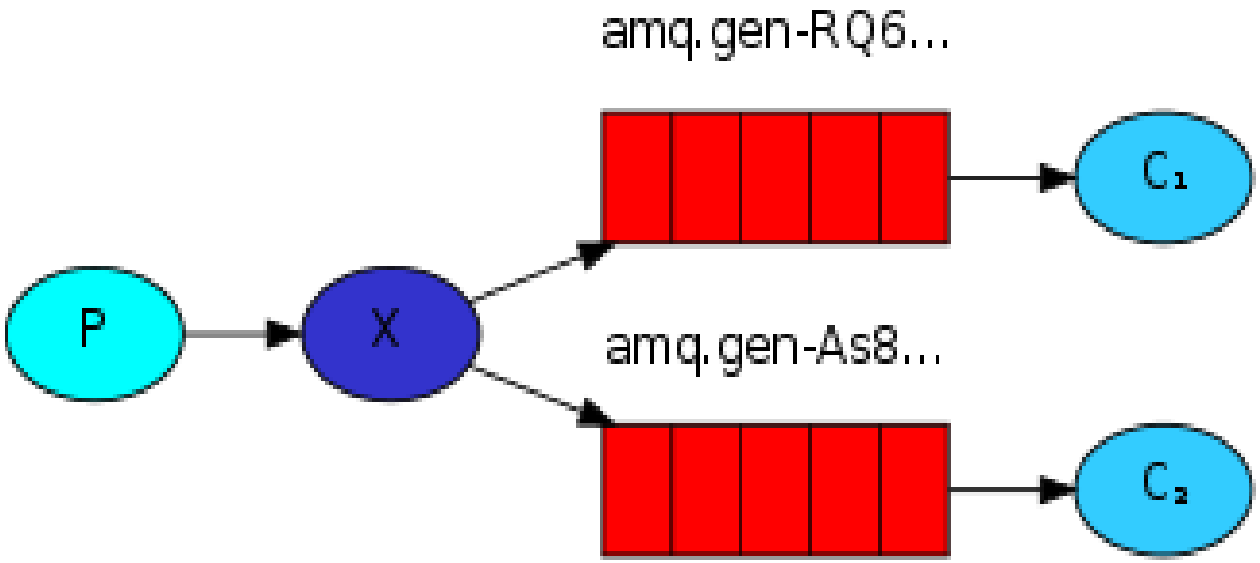
An example with an exchange : a logger

- The fanout exchange is very simple: it just broadcasts all the messages it receives to all the queues it knows.
- And that's exactly what we need for our logger.
- *channel.exchange_declare(exchange='logs', type='fanout')*

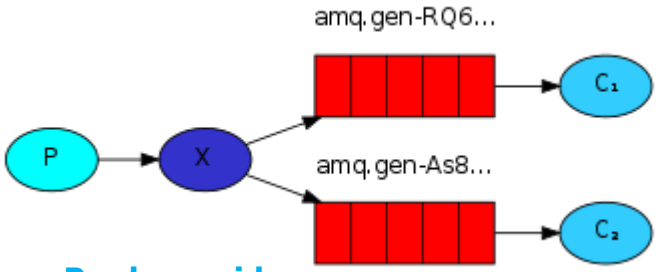
An example with an exchange : a logger

- A queue is needed for each worker to store the log messages
- How to create them when we don’t know the number of workers ? Nor their names ?
- Solution : temporary queues which are created with no name
- Note that the queue must disappear when the worker terminates (set with the *exclusive* flag)
- result = channel.queue_declare() or
- result = channel.queue_declare(exclusive=True)

An example with an exchange : a logger



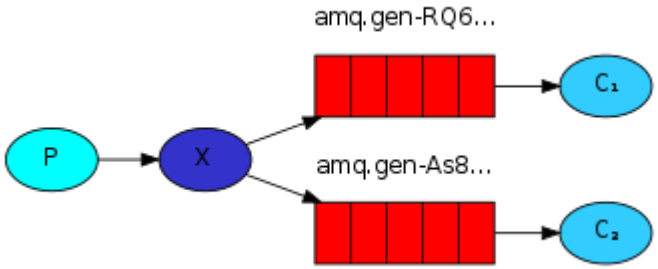
An example with an exchange : a logger



Producer side:

```
#!/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()
```

An example with an exchange : a logger



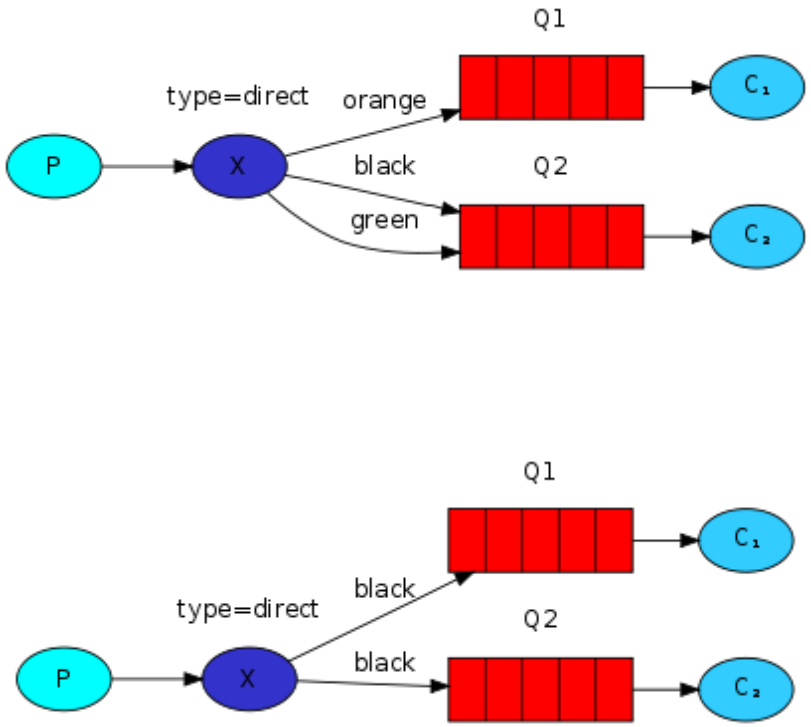
Consumer side:

```
Connection = pika.BlockingConnection(
      pika.ConnectionParameters( host='localhost'))
channel = connection.channel()
channel.exchange_declare(exchange='logs', type='fanout')

result = channel.queue_declare(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(callback, queue=queue_name,
                      no_ack=True)

channel.start_consuming()
```

A routing example

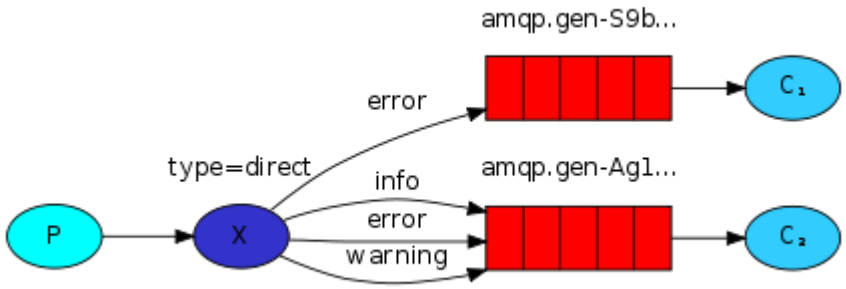
- The classic binding is : channel.queue_bind(exchange=exchange_name, queue=queue_name)
- It is possible to add a routing key and make it possible to subscribe only to a subset of the messages: channel.queue_bind(exchange=exchange_name, queue=queue_name, routing_key='black')

A routing example

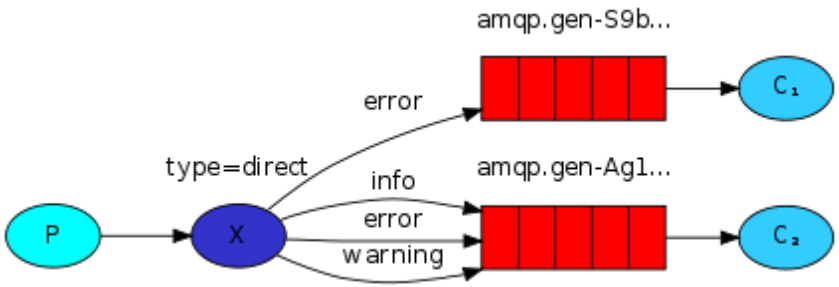


A routing example

- We want to route messages based on the severityy of the error to the appropriate queue



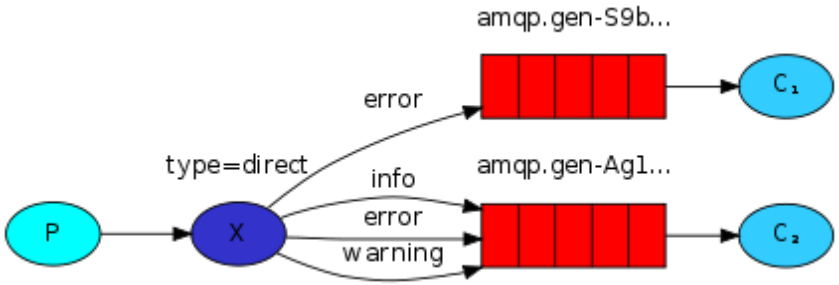
A routing example



Emit log side:

```
connection =
pika.BlockingConnection(pika.ConnectionParameters(
                host='localhost'))
channel = connection.channel()
channel.exchange_declare(exchange='direct_logs',
                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()
```

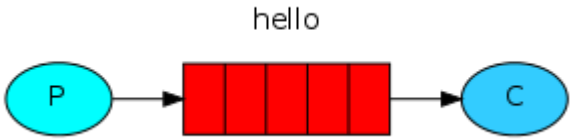
A routing example



Log reciever side:

```
connection =
pika.BlockingConnection(pika.ConnectionParameters(
        host='localhost'))
channel = connection.channel()
channel.exchange_declare(exchange='direct_logs',
        type='direct')
result = channel.queue_declare(exclusive=True)
queue_name = result.method.queue
severities = sys.argv[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(callback, queue=queue_name,
        no_ack=True)
channel.start_consuming()
```

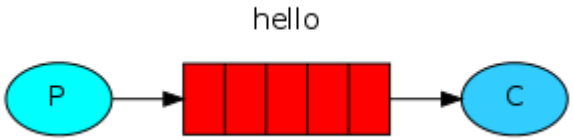
A routing example



Producer side:

```
#!/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()
```

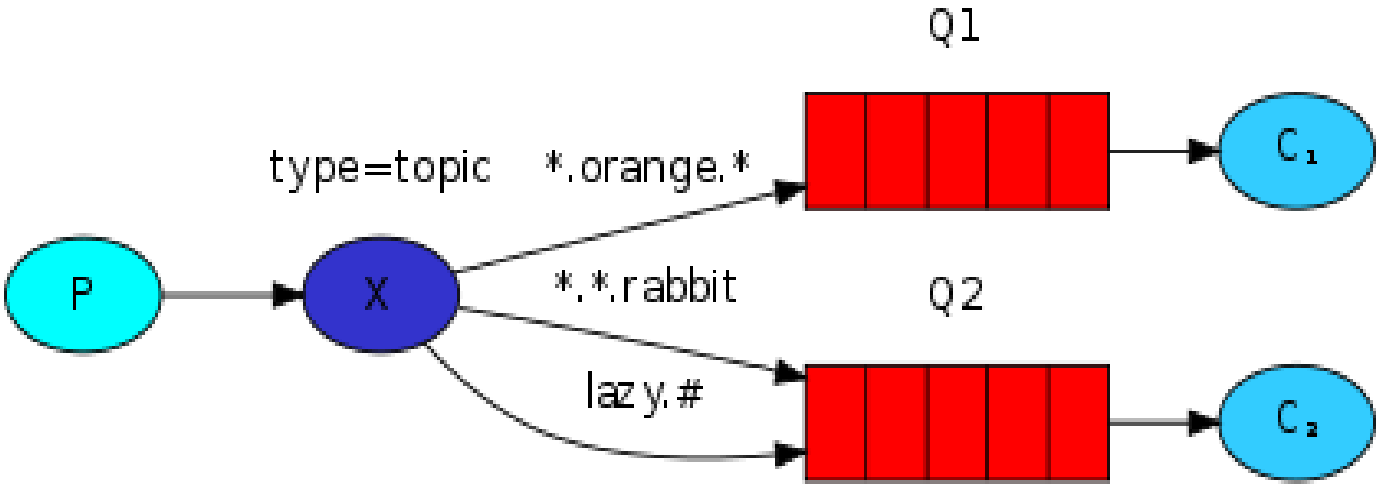
An initial example



Producer side:

```
#!/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()
```

More on routing and advanced features



Agenda

- AMQP
- Understanding messaging
- Using RabbitMQ illustrated with examples
- Administration of the broker
- Synthesis

Administration du broker

- rabbitmqctl status
- rabbitmqctl add_user yvon monpasswd
- rabbitmqctl delete_user cashing-tier
- rabbitmqctl list_users
- rabbitmqctl list_queues
- rabbitmqctl list_queues name durable auto_delete
- rabbitmqctl list_exchanges
- rabbitmqctl list_bindings
- Et plein d'autres commandes de configuration et de requêtes …

Agenda

- AMQP
- Understanding messaging
- Using RabbitMQ illustrated with examples
- Administration of the broker
- Synthesis

Synthesis

- AMQP is a powerful protocol for connecting devices and applications through various networks
- A very large set of services with enhanced functionnalities (eg, security, reliability, persitence)
- RabbitMQ is an efficient implementation that is used in IoT, services within the cloud (eg; OpenStack) and operationnal applications