1 Présentation AMQP

Voir: https://www.rabbitmq.com/tutorials/amqp-concepts.html

1.1 Notions de base

Server AMQP : Broker ou severless
Une messagerie AMQP peut être

- ⇨ **serverless** : la messagerie fonctionne alors en peer to peer. La Connection est tout de même nécessaire pour entré dans le domaine.
- ⇨ Avec **Broker** : Process (ou sous-système) supportant toutes la messagerie : toutes les requêtes arrivent dans le broker, celui-ci distribue les messages vers les clients concernés.

Virtual-host
Un broker AMQP peut supporter plusieurs 'virtual-host' : cela permet de partitionner les domaines gérés par un serveur AMQP. (saia-ve : typiquement un virtual-host par contrat)

Connexion
Un micro service se connecte une fois, il peut préciser le 'virtual-host' sur lequel il doit travailler

Channel
Lien TCP entre un client et le serveur AMQP.
Une connexion peut supporter plusieurs channel, cela permet de définir des niveaux d'urgence . Par exemple :

- Un channel pour le fond de messages (gros débit, peut critique),
- Un channel pour les messages urgents (faible débit, latence critique).

Transaction
Le concept de transaction permet à un émetteur de message de s'assurer que tous les messages à émettre sont bien émis de manière intègre : tous les messages sont émis (commit) ou-bien aucun message n'est émis (rollback).
Cela ne présuppose pas que tout les messages soient bien traités par au moins un consommateur (cela peut être assuré si entités d'échanges AMQP (exchange, queue) sont crée en QOS 'persistant') ...

Acquittement
Le receveur d'un message doit acquitter tous messages reçus. Il peut acquitter dès réception, ou après traitement.
Si le **Broker** détecte qu'un micro service est hors-service (connexion fermée ou processus tué ou socket du Channel fermé), il réémettra les messages non-acquittés sur la queue concerné.
L'ensemble transaction+Acquitement permet de fiabiliser les échanges, sans pour autans garantir le caractère ACID des transactions.

1.3 Model

Le modèle pour faire de la messagerie est un peu complexe. Cette complexité a pour but de permettre l'usage de pattern de communication divers et varié : Au lieu de définir un Subscriber et un Publisher, AMQP définie des entités diverses : Exchange, Queue, Rule, et Message.

Les différentes combinaisons de ces éléments permettent :

- Publish / subscribe : communication par abonnement / diffusion
- Job sheduling : partage de charge par requête distribué
- Broadcast / multicast : émission systématique de message à tout le monde ou quelques uns
- Rendez-vous : synchroniser deux services sur une ressource commune
- RPC : Question/reponse, synchrone et/ou asynchrone

On a donc :

- **Exchange** : défini une entrée de message dans la messagerie : il est définie par un nom, une structure de message/schema (?), une QOS (durabilité et/ou auto-delete, suplante une QOS par message).
  Un Exchange ne définie pas de queue : il s'agit d'une porte d'entrée de message.
  Pour de petites applications, on peut ne pas déclarer d'Exchange : un Exchange par défaut préexiste systématiquement (exchange de nom ''), les rules peuvent binder cet exchange avec un ensemble de queue.

- **Message** : un message (texte/binaire,'payload'), auquel on associe des méta-data ('header', à la http). Un message est émis vers un Exchange. Un header obligatoire est le « routing-key » : String sur lequel des rules peuvent appliquer du pattern-matching pour selectionner les destinataire(s) d'un message (queue).
  Les 4 méta-données suivantes sont les plus courantes :

delivery_mode	Persistent ou transcient , QOS associé individuellement à un message (peut etre suplante par le QOS de l'exchange)
content_type	mime-type ➔ 'application/json'
reply_to	QR : Nom de la queue de réponse. (RPC, rendez-vous…)
correlation_id	QR : ID message de la requête, rappelé dans le message réponse .

- **Queue** : pile de messages, typiquement une pile par consommateur.
  Accessoirement, une queue peut être utilisé pour le partage de charge : une pile, plusieurs thread/process consommateurs, dans ce cas un message représente une demande de travail, qui ne sera pris en compte que par un seul consommateur (le premier disponible).
  Un queue ne peut recevoir de message que via un Exchange (si une Rule le permet).
- **Rule** : **binding** permettant de relie un **Exchange** à une **Queue**. Pour chaque message émis sur un Exchange, la/les rules permettent de décider si le message doit être cloné/empilé dans sa Queue associée. La décision utilise les meta-data du message (conditions logique et pattern matching)
  Une Rule ne concerne qu'un exchange et une Queue, Plusieurs Rules peuvent relier un Exchange et une Queue (permet le OU de Rule),
  Une queue peut être 'binder' avec plusieurs exchanges : cela permettra à un consommateur de n'avoir qu'un point d'entrée pour tous les messages (topics) en input.

Donc :

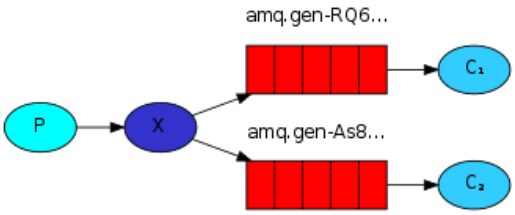
- ⇨ Les producteurs de message émettent des messages vers des **Exchange**, en documentant chaque message par un **header**,
- ⇨ Les consommateurs reçoivent des messages via des queues,
- ⇨ Le lien entre **Exchange** et **Queue** est déclaratif, via une ou plusieurs **Rules** (déclarées lors de la création d'une **Queue**) declarant un lien logique en les données d'un header de message et une **queue**

Le design des méta-data des **messages** détermine les **rules** utilisables, qui déterminent les **Exchange** à configurer. Cela effectué, chaque consommateur créera sa/ses queue(s) avec les rules associés.

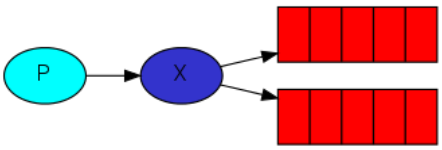
1.3.1 Topologies typiques

A chaque type de topogie correspond un type d'exchange.

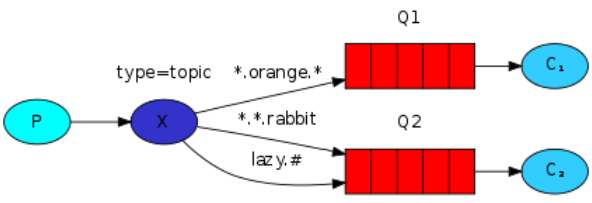
Direct Exchange Routing :
La routing-key porte le nom de la queue destinatrice : l'émetteur d'un message connait exactement son destinataire, et il est unique



Fanout exchange (one to many: broadcast)
L'Exchange envoie les messages reçus sur chaque queue qui lui est associée.



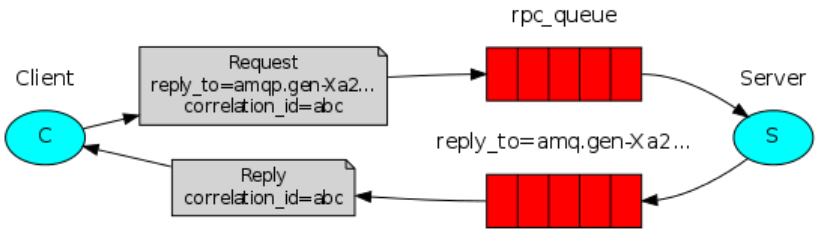
Publish/Subscribe
Exchange de type 'topic'. ON remarquera que le subscriber ne peut specifier ses besoins que par une expression reguliere sur un routing-key de message.



Header Exchange.
Direct exchange, mais toutes les meta-data peuvent être utilisés, (integer, float, boolean…)

1.4 Requête/Réponse

Exchange de type Question/réponse. La queue de réponse n'est pas encapsulée par l'API AMQP !



Le client doit créer une queue, réserver à toutes les réponses des QR qu'il fera. A l'émission de la requête, il positionne le nom de sa queue de réponse et un identifiant (correlation_id) permettant de recoller une réponse a une question (Un micro service peut être multi-thread, donc il peut y avoir plusieurs QR en parallèle).
Cela implique qu'en lecture de queue, un micro-service peut spécifier un message a dépiler, par une meta-donné, et peut attendre en bloquante (synchrone) ou en callback (asynchrone).

La réponse peut être traitée de manière synchrone (pull de la queue de reponse) ou asynchrone (callback sur queue).

On remarque que le message réponse ne passe pas par un Exchange : il est empilé directement dans la queue de réponse (reply_to) avec l'id de la requête (correlation_id).

1.5 Orchestration

Pour faire fonctionner une appli micro service orientée message, il faut un service permettant :

- De lancer /arrêter le middleware, de surveiller sa bonne marche, de le tuer/relancer si besoin
- De gérer une liste de micro service 'résident', avec leurs paramètres
- De lancer la liste des micro service configurés
- D'arrêter l'application : micro service et middleware
- Supporter une IHM pour maintenir l'appli :
    - ajout de micro-service,
    - arrêt / relance,
    - etat : stats par mq , etat du service , logs, ping…
    - test ….

IL serait dommageable de développer cela pour nos besoins.
A voir si on trouve ce qu'il faut en logiciel libre … ☺

1.6 Exemple de code API AMQP

Un producteur :

```
import pika, os, logging
logging.basicConfig()

# Parse CLODUAMQP URL (fallback to localhost)
url = os.environ.get('CLOUDAMQP URL', 'amqp://guest:guest@localhost/%2f')
params = pika.URLParameters(url)
params.socket_timeout = 5

connection = pika.BlockingConnection(params) # Connect to CloudAMQP
channel = connection.channel() # start a channel
channel.queue_declare(queue='pdfprocess') # Declare a queue
# send a message

channel.basic_publish(exchange='', routing_key='pdfprocess', body='User information')
print ("[x] Message sent to consumer")
connection.close()
```

Exemple d'un consommateur :

```
def pdf process function(msg):
  print(" PDF processing")
  print(" Received %r" % msg)

  time.sleep(5) # delays for 5 seconds
  print(" PDF processing finished");
  return;

################## Connection

url = os.environ.get('CLOUDAMQP_URL', 'amqp://guest:guest@localhost:5672/%2f')
params = pika.URLParameters(url)
connection = pika.BlockingConnection(params)
channel = connection.channel() # start a channel
channel.queue_declare(queue='pdfprocess') # Declare a queue

######### create a callback for receive message

def callback(ch, method, properties, body):
  . . . . .(body)

############## Subscibe
channel.basic_consume(callback,
  queue='pdfprocess',
  no ack=True)

########## start consuming (blocks)
channel.start_consuming()
sleep
connection.close()
```