RABBITMQ

L'épine dorsale des architectures distribuées et asynchrones d'entreprise.

MESSAGE BROKER • AMQP STANDARD • HIGH PERFORMANCE

01. Fondations & Concepts

Comprendre le paradigme de la communication orientée messages.

LE PROBLÈME DU COUPLAGE FORT

Architecture Synchrone

Chaque requête HTTP dépend de la disponibilité immédiate du service récepteur.

⚠ Risque d'effet cascade en cas de panne d'un seul service.

Architecture Asynchrone

Les services communiquent en envoyant des messages sans attendre de réponse immédiate.

✓ Robustesse, scalabilité temporelle et isolation totale des services.

ANATOMIE D'UN MESSAGE AMQP

L'Enveloppe

Contient les métadonnées de routage indispensables, comme la clé de routage (Routing Key) et le nom de l'échange cible.

Les En-têtes

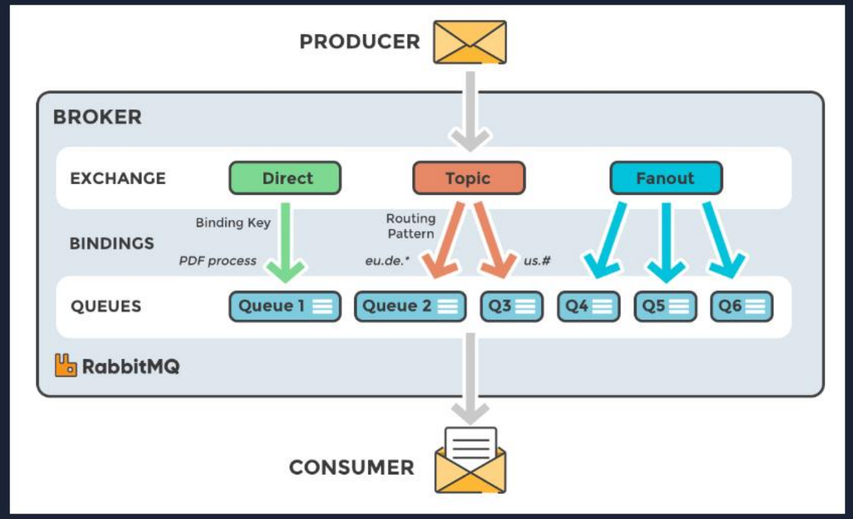
Métadonnées optionnelles (Content-Type, ID de corrélation, priorité) exploitables pour le filtrage ou le traçage applicatif.

Le Corps (Payload)

La donnée brute transmise (généralement du JSON ou du binaire) totalement opaque pour le broker RabbitMQ.

LE FLUX DE MESSAGE STANDARD

- **Producteur :** Envoie le message à un Exchange.
- **Exchange :** Oriente le message selon sa configuration.
- **Binding :** Associe l'Exchange à une Queue via des règles.
- **Queue :** Stocke le message de manière sécurisée.
- **Consommateur :** Récupère et traite le message.



EXCHANGES : DIRECT VS FANOUT

Exchange Direct

Route les messages vers les files d'attente en fonction d'une égalité stricte entre la clé de routage du message et celle du binding.

✣ Idéal pour le ciblage unitaire précis.

Exchange Fanout

Duplique et diffuse le message reçu vers toutes les files d'attente liées, en ignorant totalement la clé de routage.

📡 Idéal pour les patterns Publish-Subscribe.

EXCHANGES : TOPIC VS HEADERS

Exchange Topic

Effectue un routage dynamique basé sur des jokers : '*' (remplace exactement un mot) et '#' (remplace zéro ou plusieurs mots).

Parfait pour le routage de logs catégorisés.

Exchange Headers

Utilise la table d'en-têtes du message plutôt que la clé de routage. Plus lent mais extrêmement puissant pour le filtrage complexe.

Routage multi-critères applicatifs.

02. Fiabilité & Garanties

Assurer le transport d'informations zéro perte dans un monde instable.

ACQUITTEMENT DE MESSAGE (ACK)

- **Acknowledge (Ack) :** Le consommateur notifie RabbitMQ du traitement réussi. Le message est alors supprimé de la file en toute sécurité.
- **Negative Ack (Nack) :** Signale un échec de traitement. Le message peut être réintroduit en tête de file (Requeue) ou rejeté définitivement.
- **Durabilité (Durable) :** Configuration des files et messages pour persister sur le disque dur, résistant ainsi aux redémarrages de serveurs.

DEAD LETTER EXCHANGES (DLX)

Messages expirés

Les messages dont la durée de vie (TTL) est dépassée sont automatiquement redirigés vers un DLX d'archivage.

Messages rejetés

Les messages provoquant des erreurs systématiques d'analyse (Nack sans requeue) sont isolés pour analyse.

Queue saturée

Si la file d'attente atteint sa taille limite, les messages les plus anciens sont évacués vers le DLX de secours.

03. Installation & Linux

Déployer RabbitMQ sur un serveur de production Linux.

INSTALLATION DE RABBITMQ LINUX

RabbitMQ s'exécute sur la machine virtuelle **Erlang (BEAM)**. Il est crucial d'installer des versions parfaitement compatibles.

Commandes d'installation de base sur Debian/Ubuntu :

```
# Installation de Erlang & RabbitMQ sudo apt-get update sudo apt-get install -y erlang-nox sudo apt-get install -y rabbitmq-server
```

Pour administrer RabbitMQ en production, l'activation des outils CLI et de l'interface d'administration est indispensable :

```
# Activer le plugin de Management sudo rabbitmq-plugins enable rabbitmq_management # Ajouter un utilisateur administrateur sudo rabbitmqctl add_user admin MonMotDePasse sudo rabbitmqctl set_user_tags admin administrator sudo rabbitmqctl set_permissions -p "/" admin ".*" ".*" ".*"
```

CONFIGURATION SYSTEMD LINUX

Pour s'assurer que le service démarre automatiquement après une panne système, on s'appuie sur le gestionnaire de services **systemd** :

```
# Vérifier l'état du service sudo systemctl status rabbitmq-server # Activer au démarrage de la machine sudo systemctl enable rabbitmq-server # Redémarrer après changement de conf sudo systemctl restart rabbitmq-server
```

Gestion des limites système Linux importantes (`/etc/security/limits.conf`) pour les sockets ouverts (File Descriptors) :

```
# Limite d'ouverture de fichiers minimale rabbitmq soft nofile 65536 rabbitmq hard nofile 65536
```

**Une limite trop basse (par défaut 1024) empêchera RabbitMQ d'accepter plus de 1000 connexions concurrentes.*

04. Développement & Code

Intégrer RabbitMQ dans vos différentes couches de services logicielles.

CLIENT : PYTHON & NODE.JS

Python (Librairie pika)

```
import pika conn = pika.BlockingConnection( pika.ConnectionParameters('localhost') ) channel = conn.channel() channel.queue_declare(queue='task_queue') # Envoi du message channel.basic_publish( exchange='', routing_key='task_queue', body='Hello World!' ) conn.close()
```

Node.js (amqplib)

```
const amqp = require('amqplib'); async function send() { const conn = await amqp.connect('amqp://localhost'); const ch = await conn.createChannel(); const q = 'task_queue'; await ch.assertQueue(q, {durable: true}); ch.sendToQueue(q, Buffer.from('Hello!')); setTimeout(() => { conn.close(); }, 500); } send();
```

CLIENT : JAVA & PHP

Java (Spring AMQP)

```
@Configuration public class RabbitConfig { @Bean public Queue queue() { return new Queue("task_queue", true); } } // Envoi du message dans un service @Autowired private RabbitTemplate template; public void send(String msg) { template.convertAndSend("task_queue", msg); }
```

PHP (php-amqplib)

```
use PhpAmqpLib\Connection\AMQPStreamConnection; use PhpAmqpLib\Message\AMQPMessage; $connection = new AMQPStreamConnection( 'localhost', 5672, 'guest', 'guest' ); $channel = $connection->channel(); $channel->queue_declare('task_queue', false, true); $msg = new AMQPMessage('Hello PHP!'); $channel->basic_publish($msg, '', 'task_queue');
```

05. Production & HA

Dimensionner et surveiller son infrastructure RabbitMQ en production.

CLUSTERING & QUORUM QUEUES

3+

NŒUDS MINIMAUX

Tolérance aux Pannes Avancée

Le clustering natif de RabbitMQ regroupe plusieurs serveurs physiques en un seul broker logique.

Les **Quorum Queues** implémentent l'algorithme de consensus **Raft** pour répliquer de manière stricte et sécurisée les messages sur la majorité des nœuds.

Fin des "Split-Brain" de réseaux.

COMPARATIF : BROKER ET CACHE

Caractéristique	RabbitMQ	Apache Kafka	Redis Cache
Pattern Principal	Message Broker (AMQP)	Event Streaming (Log append)	Key-Value Store / Pub-Sub léger
Persistance	Durable (jusqu'à consommation)	Permanente (selon rétention)	En mémoire (option disque)
Débit d'écriture	Élevé (~10-100k/s par file)	Extrême (Plusieurs M/s)	Ultra Élevé (Microsecondes)
Routage	Ultra complexe et dynamique	Simple par clé/partition	Aucun routage avancé

ADMINISTRATION DASHBOARD

L'interface web intégrée de RabbitMQ offre une surveillance complète de l'activité en temps réel :

- Débits d'écriture et de lecture par seconde.
- Connexions clientes et canaux ouverts.
- Consommation RAM et espace disque utilisé.

Accessible par défaut sur le port 15672.

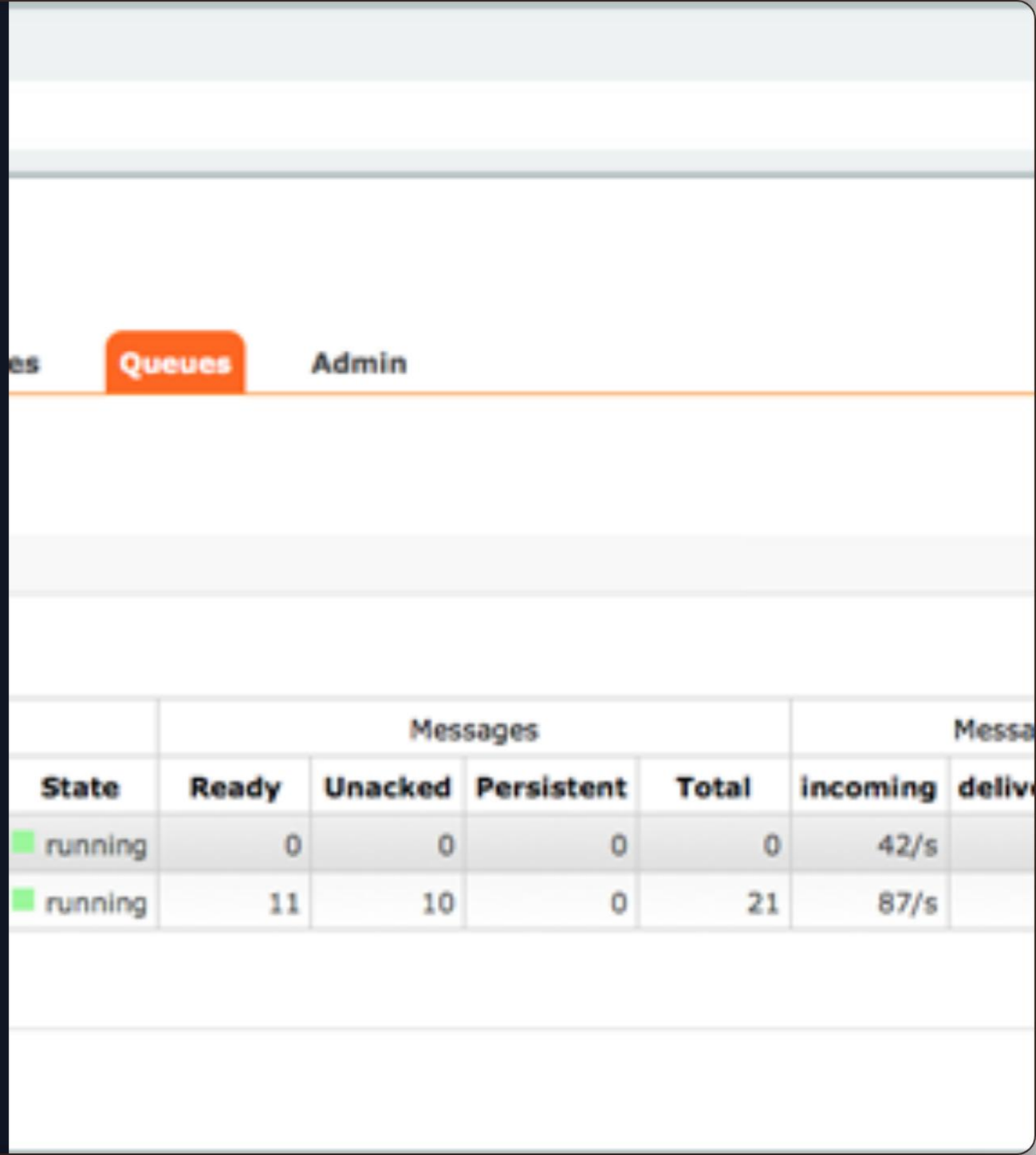


IMAGE SOURCES

https://www.cloudamqp.com/img/blog/exchanges-topic-fanout-direct.png

Source: www.cloudamqp.com

https://www.cloudamqp.com/img/blog/queues.png

Source: www.cloudamqp.com