



Architectures Logicielles Modernes

Séance 5 — L'asynchrone : RabbitMQ, ou l'art de ne plus mourir ensemble

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et
Techniques, Université Cheikh Anta Diop de Dakar

<https://ebtoure.sn/cours/architectures-logicielles-modernes/chapitres/05-l-asynchrone.html>

Version du 11 octobre 2026



Version en ligne

Architectures Logicielles — Séance 5

L'asynchrone : RabbitMQ, ou l'art de ne plus mourir ensemble

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

Objectifs de la séance

Contenu

Dernière séance : les événements du Lab 3 quittent la JVM et traversent le réseau via **RabbitMQ**. Les notifications consomment depuis une **queue durable** ; le module paiement déménage en **Python (pika)** — deuxième extraction du cours, cette fois *asynchrone* : sans façade, sans timeout, sans 503. Et la démonstration miroir de la panne du Lab 4 : un consommateur éteint, ce sont des messages **qui attendent** — pas une panne.

- Comprendre le **broker** : exchange, queue durable, clé de routage, acquittement.
- Publier depuis Spring (**Spring AMQP**) et consommer depuis Python (**pika**).
- Éprouver la **résilience** : arrêt, attente, rattrapage — zéro message perdu.
- Savoir trancher, pour chaque intégration : **synchrone ou asynchrone ?**

Le bilan des blessures

Trois douleurs, soigneusement conservées

Nous avons laissé trois blessures ouvertes — volontairement :

- **La panne synchrone (Lab 4) :** *livraison-service* éteint $\Rightarrow$ la confirmation meurt en 503. Compréhensible pour l'assignation (il *faut* une réponse)... mais les SMS et le paiement, eux, n'exigeaient aucune réponse — pourquoi partager le sort de qui que ce soit ?
- **Le statut manuel :** la course atteint 100,%, le service *sait* — et pourtant quelqu'un doit cliquer « Marquer livrée ».
- **L'état qui s'évapore :** le service redémarre, les courses s'oublient, la base dit le contraire. Deux vérités, aucune réconciliation.

Le diagnostic commun

Ces trois douleurs ont la même racine : nos services ne savent communiquer qu'**en direct et à l'instant**. Il leur manque un endroit où déposer un fait pour qu'il soit traité *quand le destinataire pourra* — une **boîte aux lettres durable**.

Le broker

L'idée simple : le téléphone et le message

Deux façons de joindre quelqu'un

L'appel téléphonique : il faut que l'autre décroche, maintenant. S'il dort, téléphone éteint — échec, rappelez plus tard. **Le message WhatsApp** : vous l'envoyez et passez à autre chose. Son téléphone est éteint ? Le message *attend*. À l'allumage, il reçoit *tout*, dans l'ordre — rien ne s'est perdu.

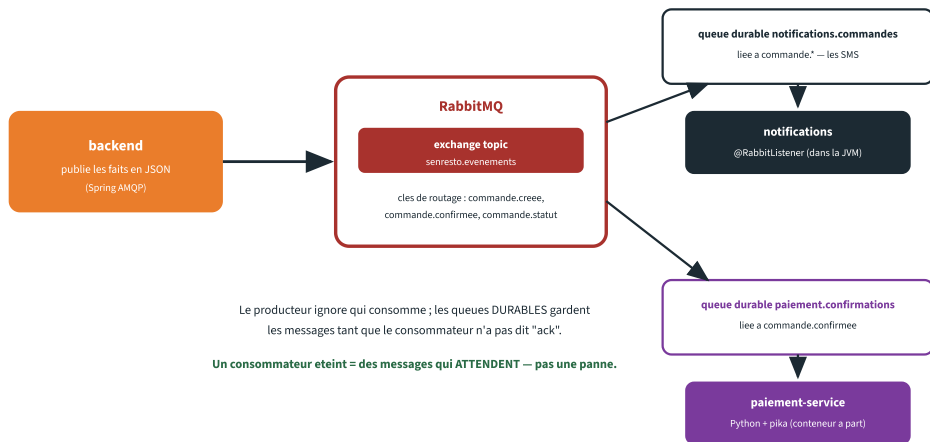
Dans notre lab, mot pour mot :

- l'appel téléphonique = l'appel HTTP du Lab 4 : il faut que **livraison-service** « décroche » — sinon, 503 ;
- le message WhatsApp = l'événement via RabbitMQ : le backend publie et **continue sa route** ;
- le téléphone éteint = **paiement-service** arrêté : les messages **attendent** dans sa queue ;
- l'allumage = le redémarrage : **rattrapage intégral**, dans l'ordre, zéro perte.

(RabbitMQ est le serveur qui garde les messages — la « boîte » entre l'envoyeur et le destinataire.)

RabbitMQ : la boîte aux lettres durable

RabbitMQ : la boîte aux lettres durable entre les services



Le trajet d'un message, pas à pas

Suivons `CommandeConfirree` de bout en bout :

Le backend le dépose au **guichet de tri** (l'*exchange* `senresto.evenements`) avec une **étiquette** (la *clé de routage*) : `commande.confirree`.

Le guichet regarde qui s'est **abonné** (les *bindings*) : la boîte des notifications a demandé « tout `commande.*` » ; celle du paiement, seulement `commande.confirree`.

Le message est **copié** dans chaque boîte concernée (les *queues*) — une copie par abonné, chacun la sienne.

Chaque consommateur relève **sa** boîte, à son rythme — réveillé ou pas au moment de l'envoi, peu importe.

À retenir

Le backend ne connaît que le guichet et l'étiquette. Il ignore **qui** est abonné, **combien** ils sont, et **s'ils sont réveillés**. C'est le découplage du Lab 3 — étendu au réseau et au temps.

Qui reçoit quoi ? Le routage en pratique

Nos trois étiquettes face à nos deux boîtes :

Clé du message	notifications	paiement
	(abonnée à <code>commande.*</code>)	(abonnée à <code>commande.confirmer</code>)
<code>commande.creee</code>	reçoit	—
<code>commande.confirmer</code>	reçoit	reçoit
<code>commande.statut</code>	reçoit	—

- Le `*` de l'abonnement remplace **un mot** : `commande.*` attrape les trois clés.
- Demain, un module statistiques s'abonne à `commande.#` : **zéro modification** chez le backend — comme le module paiement du Lab 3, version réseau.

Le vocabulaire qui suffit

Exchange	le guichet de tri : reçoit chaque message et le route
Clé de routage	l'étiquette du message (<code>commande.confirmer</code>)
Queue durable	la boîte d'un consommateur ; survit aux redémarrages
Binding	l'abonnement d'une queue (<code>commande.*</code> = tout)
Ack	l'accusé du consommateur ; sans lui, le message reste

À retenir

Cinq mots, et vous savez lire l'UI de gestion de RabbitMQ — votre poste d'observation de tout le lab. La garantie qu'ils construisent ensemble arrive à la slide suivante.

L'ack : pourquoi APRÈS le traitement ?

L'*ack* est l'accusé de réception du consommateur. Tout tient dans **le moment** où on l'envoie. Deux scénarios de crash :

- **Ack *avant* de traiter** : le consommateur acquitte, la queue efface le message... et le consommateur crashe avant d'avoir traité. Le message est **perdu pour toujours** — un encaissement envolé.
- **Ack *après* avoir traité** (notre règle) : le consommateur crashe en plein traitement ? Pas d'ack reçu — la queue **redonne le message** au redémarrage. Rien ne se perd... mais le message peut être **livré deux fois**.

La garantie *at-least-once*

Queue durable + message persistant + ack après traitement = chaque message est livré **au moins une fois**, à travers pannes et redémarrages. « Au moins » — donc *parfois deux fois* : c'est un choix, pas un défaut. L'alternative (au plus une fois) accepte de **perdre** des messages — inacceptable pour un paiement.

Publier et consommer

Publier : le pont vers le broker

Nos modules publient déjà des faits (Lab 3). Un **pont** suffit — il écoute les événements internes et les réexpédie en JSON :

```
@Component
class PontAmpq {
    private final RabbitTemplate rabbit;
    // ...
    @EventListener
    void quandCommandeConfirmer(CommandeConfirmer evt) {
        rabbit.convertAndSend("senresto.evenements", // exchange
                             "commande.confirmer",   // cle de routage
                             evt);                  // serialise en JSON
    }
}
```

- **CommandeMetier** n'a **pas bougé** d'une ligne : il publie ses faits comme au Lab 3, sans savoir qu'un pont les fait désormais voyager. Les frontières du modolith paient une dernière fois.

Consommer côté Java : notifications

Côté monolithe, notifications troque son `@EventListener` contre une queue durable — le Strangler, encore : l'ancien chemin reste sous `@Profile("!amqp")`, le nouveau vit sous `@Profile("amqp")` :

```
@RabbitListener(queues = "notifications.commandes")
void surEvenement(Map<String, Object> evt,
    @Header("amqp_receivedRoutingKey") String cle) {
    // memes textes de SMS qu'avant, aiguilles par la cle
}
```

- La queue est déclarée **durable** : elle survit aux redémarrages — du broker comme du consommateur.
- Ce qui change pour l'utilisateur : **rien**. Mêmes SMS, même ordre. Ce qui change pour l'architecte : les SMS ne peuvent **plus se perdre**.

Consommer côté Python : le paiement déménagement

Le paiement quitte le monolithe pour son propre conteneur — **pika**, le dialecte AMQP de Python :

```
def sur_message(canal, methode, proprietes, corps):
    evt = json.loads(corps)
    print(f"[PAIEMENT] Encaissement mobile money simule : "
          f"{evt['montantTotal']} FCFA pour {evt['numero']}")
    canal.basic_ack(methode.delivery_tag)  # l'ack APRES le traitement

canal.basic_consume(queue="paiement.confirmations", on_message_callback=sur_message)
```

La deuxième extraction — comparez au Lab 4 !

Extraire `paiement` n'a demandé **ni façade, ni timeout, ni 503** : personne n'attend sa réponse. L'extraction asynchrone est *radicalement* moins chère — quand la nature de l'échange le permet.

La resilience

La démonstration miroir

Lab 4 : appel SYNCHRONES en panne



il FAUT une réponse maintenant :
timeout après 2 s, puis

503 — la confirmation EST MORTE

couplage temporel : les deux services
doivent être vivants au même instant

Lab 5 : événement ASYNCHRONES en panne



personne n'attend de réponse :
les confirmations REUSSISSENT,

les messages ATTENDENT dans la queue

au redémarrage : rattrapage intégral,
[PAIEMENT] x3 — zero message perdu

La règle de décision du semestre

Il FAUT une réponse pour continuer (assignation, ETA) : appel synchrone — et on assume le couplage temporel.
On INFORME d'un fait (SMS, paiement, stats...) : événement asynchrone — et les pannes deviennent des retards.

La même panne, deux destins : c'est le choix d'intégration qui décide, pas la chance.

Le message qui revient : un scénario concret

Rejouons le crash de la slide « ack » avec notre paiement :

`paiement-service` reçoit `CommandeConfirmer` pour CMD-2026-0042, affiche `[PAIEMENT] Encaissement... 3\,500 FCFA...`

...et **crashe** juste avant d'envoyer l'ack (coupure, OOM, redéploiement — la vie).

La queue n'a pas reçu d'ack : au redémarrage, elle **redonne le même message**.

Sans précaution : **deuxième encaissement** de CMD-2026-0042. Le client paie deux fois son mafé.

Le point important

Ce n'est pas un bug de RabbitMQ : c'est la conséquence *logique* du choix « ne jamais perdre » (at-least-once). La question n'est donc pas « est-ce que ça arrivera ? » mais « **que fait mon code quand ça arrivera ?** » — réponse à la slide suivante.

L'idempotence : « au moins une fois » se paie

Définition

Un traitement est **idempotent** si le rejouer ne change rien : traiter deux fois le même message = le traiter une fois. C'est le prix de l'*at-least-once* — et la règle d'or de tout webhook de paiement réel (Wave, Orange Money...).

- Recette du lab : chaque événement porte un **identifiant** ; le consommateur garde les identifiants déjà traités et **ignore les revenants** — en loggant [PAIEMENT] *deja traite, ignore*.
- Au lab (bonus) : un **webhook mobile money simulé** rejoué deux fois — un seul encaissement. Encaisser deux fois un client, c'est le perdre trois fois.

L'idempotence, ce qu'il faut en retenir

À retenir

L'idempotence n'est pas une option avancée : c'est le **compagnon obligatoire** de l'at-least-once. Chez nous : un **set** des identifiants traités. En production : une table dédiée, ou une contrainte d'unicité en base.

Le lab et le cours

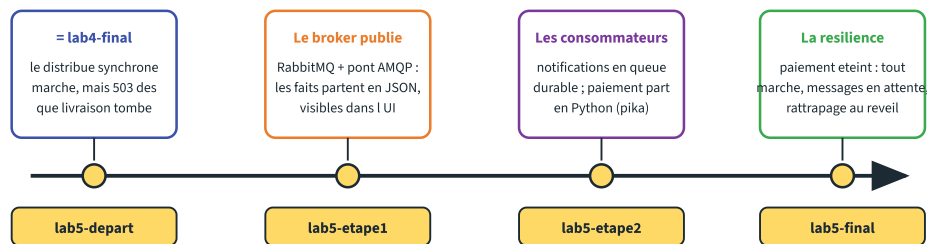
L'orchestre final : six conteneurs, trois langages

L'état du système à la fin du semestre — chaque brique a été ajoutée pour une raison précise :

Conteneur	Techno	Pourquoi il existe
<code>backend</code>	Spring Boot	le monolithe modulaire : catalogue, commandes, notifications
<code>livraison-service</code>	Python FastAPI	extraction <i>synchrone</i> (Lab 4) : écosystème géo, déploiement à part
<code>paiement-service</code>	Python pika	extraction <i>asynchrone</i> (Lab 5) : sans façade ni 503
<code>rabbitmq</code>	Erlang	la boîte aux lettres durable entre les trois
<code>postgres</code>	—	les données du monolithe
<code>frontend</code>	React	l'interface — et la carte qui a tout révélé

Trois langages qui coopèrent sans se connaître : par contrats HTTP et par messages — jamais par partage de code.

Le chemin du Lab 5



Bloques ? git checkout du tag suivant : personne ne reste au bord de la route.

Départ : votre **lab4-final**. Rôles : l'Appariteur pilote **paiement-service** (pika) ; le Notaire les queues et bindings ; l'Huissier compose + RabbitMQ ; le Rédacteur l'UI de gestion et la preuve du rattrapage ; le Doyen la démo finale et l'**ADR n°5** : « **Synchrone ou asynchrone ?** » — le tableau de décision de *toutes* vos intégrations.

L'essentiel — et l'arc du semestre

À retenir de la séance

Un **broker** transforme les pannes en retards : queue durable + ack = *at-least-once*, dont le prix est l'**idempotence**. La règle de décision : il faut une réponse $\Rightarrow$ synchrone (et l'on assume le couplage temporel) ; on informe d'un fait $\Rightarrow$ asynchrone (et les consommateurs vivent leur vie).

Le chemin parcouru en cinq séances

Un God controller qui souffrait (S1) $\rightarrow$ des couches qui séparent la technique (S1) $\rightarrow$ un hexagone qui protège le métier (S2) $\rightarrow$ des modules aux frontières gardées par un test (S3) $\rightarrow$ un premier microservice, extrait avec précaution et payé au prix du synchrone (S4) $\rightarrow$ des événements qui voyagent, et des services qui ne meurent plus ensemble (S5). Vous n'avez pas appris *des* architectures : vous avez appris à **faire évoluer** une architecture — sous tests, par étapes, chaque décision tracée dans un ADR. C'est exactement le métier.

Ressources de la séance

- **TP** · [Lab 5 — L'asynchrone avec RabbitMQ](#) (455 Ko)