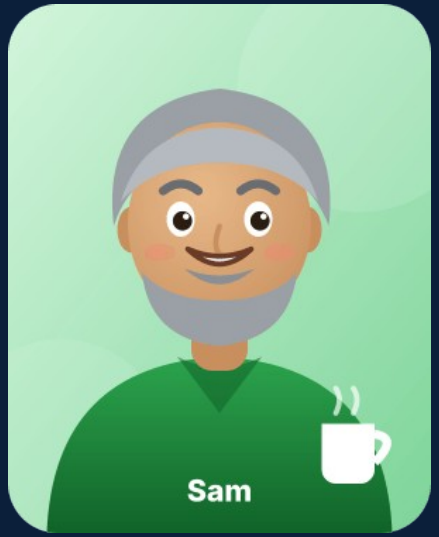
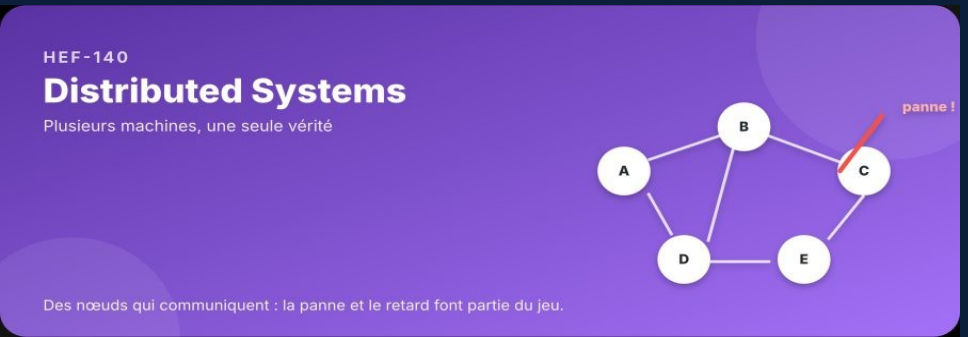
HEF-140 · MISSION 03 / 10

Latence, timeout et retry

Un retry mal placé transforme une panne locale en catastrophe globale. Bien utilisé, il masque les pannes brèves. La différence tient à quelques chiffres.





Badge : Maître des délais

105 min · avancé

L'histoire

Un mardi, la base de données de HashBox ralentit pendant 30 secondes. Aucun problème grave en soi. Mais la passerelle réessaie 3 fois, le service d'inscription réessaie 3 fois, et la couche d'accès à la base réessaie 3 fois. Chaque requête utilisateur devient 4 × 4 × 4 = **64 requêtes** vers une base déjà en difficulté. La base ne se remet pas. La panne dure 40 minutes.



> *« Chaque maillon a fait quelque chose de raisonnable, dit Sam. Réessayer. Ensemble, ils ont fabriqué une avalanche. En distribué, la sûreté est globale : il faut un budget de retry, des échéances partagées, et parfois une deuxième requête… exprès, pour réduire la queue de latence. »*

Tu apprends à calculer l'amplification des retries, à propager des échéances, à utiliser le hedging contre la latence de queue, et à plafonner les retries avec un budget.

Ta mission

À la fin, je serai capable de…

1. calculer l'amplification de charge due aux retries en couches
2. propager une échéance (deadline) à travers une chaîne d'appels
3. expliquer et simuler les requêtes doublées (hedged requests) contre la latence de queue
4. implémenter un budget de retries pour éviter les tempêtes
5. choisir des délais d'attente à partir de percentiles mesurés

Défi d'échauffement

Défi d'échauffement

Une file au cinéma : si chaque client lance trois personnes de plus dans la file « au cas où » quand elle avance lentement, que devient la file ? Cite un cas de ta vie où essayer à nouveau, plusieurs à la fois, a aggravé la situation (réseau saturé, appels en pic, serveur de réservation).

Le concept

La latence en distribué

Une requête utilisateur traverse souvent N services. Si chacun a une latence p99 de 100 ms et qu'une requête en appelle 10 en séquence, la probabilité qu'**au moins un** soit lent est $1 - 0{,}99^{10} \approx 9{,}6\ \%$. En parallèle sur 100 services (*fan-out*), $1 - 0{,}99^{100} \approx 63\ \%$. La **latence de queue** (*tail latency*) se propage et s'amplifie.

Choisir un délai d'attente

Trop court	Trop long
fausses alertes, relances inutiles, charge accrue	ressources bloquées, utilisateurs qui attendent, cascades

Méthode : regarder la distribution réelle. Un délai vers **p99,9 de la latence normale** (ou 2 à 3 × p99) évite la plupart des faux positifs ; la requête qui l'atteint est considérée comme échouée.

Le concept

Retry : quand et comment

On réessaie **uniquement** :

- les erreurs **transitoires** (délai, 503, connexion refusée), pas les erreurs métier ;
- les opérations **idempotentes** (mission 4) ;
- avec **recul exponentiel et gigue** (HEF-120, mission 9) ;
- avec un **nombre limité** de tentatives ;
- **à un seul niveau** de la chaîne (de préférence le plus proche de l'utilisateur ou de la dépendance).

L'amplification

Le concept

L'amplification (suite)

Si chaque couche fait `r` tentatives (1 normale + `r − 1` relances) et que la dépendance échoue, le nombre d'appels à la dépendance est `r^L` pour `L` couches. Avec 3 tentatives sur 3 couches : **27 ×**. Avec 4 tentatives : 64 ×.

Pire : les retries arrivent **quand le système est déjà fragile**. Une panne brève peut devenir une **panne métastable** : même après la disparition de la cause initiale, la surcharge due aux retries empêche le retour à la normale.

Le budget de retries

Plafonner les relances à une **fraction du trafic** (par exemple 10 % des requêtes). Principe du **seau à jetons** :

- chaque requête réussie ajoute 0,1 jeton ;
- une relance consomme 1 jeton ;

Le concept

Le budget de retries (suite)

- sans jeton, on **n'essaie pas** : on échoue vite.

En panne totale, les relances s'arrêtent d'elles-mêmes après quelques-unes ; en panne partielle (1 % d'échec), elles passent. La charge supplémentaire est bornée à ~10 %.

Propager l'échéance (*deadline propagation*)

L'appelant initial dispose d'un **budget de temps** (par exemple 1 s). Chaque appel en aval reçoit l'**échéance restante**, pas un nouveau délai. Une couche qui reçoit une requête dont l'échéance est dépassée ne **commence pas** le travail : personne n'attend plus la réponse. Sans cela, on continue à travailler pour des clients déjà partis.

Les requêtes doublées (*hedging*)

Le concept

Les requêtes doublées (*hedging*) (suite)

La latence de queue est causée par des ralentissements **ponctuels** (ramasse-miettes, voisin bruyant). Un remède : si la réponse n'est pas arrivée après p95, envoyer **la même requête à un autre réplicat** et prendre la première réponse. Coût : environ 5 % de requêtes supplémentaires ; gain : un p99 divisé par 2 ou plus. Prérequis : opérations **idempotentes** (lectures) et capacité en réserve.

Autres protections

Mécanisme	Effet
Disjoncteur (HEF-120)	cesser d'appeler une dépendance malade
Délestage	refuser tôt le travail en excès
Limitation de débit	plafonner un client
Isolation (*bulkhead*)	une dépendance lente ne monopolise pas tous les threads
Priorités	traiter l'important d'abord

Le concept

Autres protections (suite)

Mécanisme	Effet
Repli	réponse dégradée plutôt qu'une erreur

🧠 ***Astuce*** *: place le retry à* ***un seul endroit****, protège-le par un* ***budget****, et fais circuler l'****échéance****. Ces trois règles évitent la plupart des avalanches.*

Exemple pas à pas

Problème : calculer et simuler l'amplification des retries, propager une échéance, mesurer l'effet du hedging sur la latence de queue, et construire un budget de retries qui empêche la tempête.

```
import math, random

# ---------------- 1. amplification des retries en couches ----------------
def amplification(couches, tentatives):
    return tentatives ** couches

def simuler_chaine(couches, tentatives, p_echec_base, graine=1, n=20000):
    # chaque requête utilisateur traverse `couches` niveaux ; chaque niveau réessa
   ie jusqu'à `tentatives` fois si l'aval échoue
    rng, appels_base = random.Random(graine), [0]
    def appeler(niveau):
        if niveau == couches:
            appels_base[0] += 1
            return rng.random() >= p_echec_base                    # True = succès
        for _ in range(tentatives):
```

Exemple pas à pas (suite)

```
            if appeler(niveau + 1): return True
        return False
    succes = sum(appeler(0) for _ in range(n))
    return appels_base[0] / n, succes / n

print("couches × tentatives | échec base | appels à la base par requête | succès u
   tilisateur")
for tentatives in (1, 3):
    for p in (0.0, 0.3, 1.0):
        charge, ok = simuler_chaine(3, tentatives, p)
        print(f"        3 × {tentatives}          |   {p:4.0%}    |          {char
   ge:6.2f}              |   {ok:6.1%}")
assert amplification(3, 3) == 27 and amplification(3, 4) == 64
assert abs(simuler_chaine(3, 3, 1.0)[0] - 27) < 0.01                  # panne total
   e : charge × 27
assert simuler_chaine(3, 3, 0.3)[1] > 0.99 > simuler_chaine(3, 1, 0.3)[1]

# ---------------- 2. propagation d'échéance ----------------
class Contexte:
```

Exemple pas à pas (suite)

```
    def __init__(self, echeance): self.echeance = echeance          # instant abso
   lu (temps simulé)

def appeler_service(nom, ctx, duree, horloge, aval=None, travail_inutile=None):
    if horloge["t"] >= ctx.echeance:
        return f"{nom} : échéance dépassée, travail NON démarré"
    horloge["t"] += duree
    if aval: return aval(ctx)
    return "ok"

def chaine(ctx, horloge, propager):
    base = appeler_service("base", ctx, 0.4, horloge)
    return base
def service(ctx, horloge, propager):
    c = ctx if propager else Contexte(horloge["t"] + 10)           # sans propagat
   ion : chaque couche reprend un délai neuf
    return appeler_service("service", c, 0.3, horloge, aval=lambda cc: chaine(cc,
   horloge, propager))
def passerelle(horloge, propager, budget=0.5):
```

Exemple pas à pas (suite)

```
    ctx = Contexte(horloge["t"] + budget)                          # l'utilisateur
    n'attend que 0,5 s
    return appeler_service("passerelle", ctx, 0.2, horloge, aval=lambda cc: servic
   e(cc, horloge, propager))

h1, h2 = {"t": 0.0}, {"t": 0.0}
r1, r2 = passerelle(h1, True), passerelle(h2, False)
print(f"avec propagation : {r1} (temps de travail utilisé : {h1['t']:.1f} s)")
print(f"sans propagation : {r2} (temps de travail utilisé : {h2['t']:.1f} s alors
   que l'utilisateur est parti depuis 0,5 s)")
assert "NON démarré" in r1 and h1["t"] < h2["t"] and r2 == "ok"

# ---------------- 3. hedging : réduire la latence de queue ----------------
def latence(rng):
    # 97 % des requêtes : ~20 ms ; 3 % : ralentissement ponctuel de ~300 ms (ramas
   se-miettes, voisin bruyant)
    base = rng.lognormvariate(math.log(0.020), 0.25)
    return base + (rng.uniform(0.2, 0.4) if rng.random() < 0.03 else 0.0)
```

Exemple pas à pas (suite)

```
def percentile(v, p):
    v = sorted(v); return v[min(len(v) - 1, math.ceil(p / 100 * len(v)) - 1)]

def simuler(hedging, seuil, n=100000, graine=3):
    rng, lat, requetes = random.Random(graine), [], 0
    for _ in range(n):
        a = latence(rng); requetes += 1
        if hedging and a > seuil:                                  # pas de répo
   nse après `seuil` : on envoie un doublon
            requetes += 1
            b = seuil + latence(rng)
            a = min(a, b)
        lat.append(a)
    return percentile(lat, 50), percentile(lat, 99), percentile(lat, 99.9), requet
   es / n

seuil = percentile([latence(random.Random(9)) for _ in range(1)] + [latence(random
   .Random(i)) for i in range(5000)], 95)
sans, avec = simuler(False, seuil), simuler(True, seuil)
```

Exemple pas à pas (suite)

```
print(f"seuil de doublage (p95) = {seuil * 1000:.0f} ms")
print(f"sans hedging : p50 {sans[0] * 1000:5.0f} ms, p99 {sans[1] * 1000:5.0f} ms,
    p99,9 {sans[2] * 1000:5.0f} ms | charge ×{sans[3]:.2f}")
print(f"avec hedging : p50 {avec[0] * 1000:5.0f} ms, p99 {avec[1] * 1000:5.0f} ms,
    p99,9 {avec[2] * 1000:5.0f} ms | charge ×{avec[3]:.2f}")
assert avec[1] < sans[1] / 3 and avec[3] < 1.10 and sans[1] > 0.2

# ---------------- 4. budget de retries ----------------
class BudgetRetries:
    def __init__(self, ratio=0.1, plafond=10):
        self.ratio, self.plafond, self.jetons = ratio, plafond, plafond
    def succes(self): self.jetons = min(self.plafond, self.jetons + self.ratio)
    def peut_reessayer(self):
        if self.jetons >= 1: self.jetons -= 1; return True
        return False

def charge_avec(strategie, p_echec, n=20000, graine=4):
    rng, appels, budget = random.Random(graine), 0, BudgetRetries()
    succes = 0
```

Exemple pas à pas (suite)

```
    for _ in range(n):
        for tentative in range(4):
            appels += 1
            if rng.random() >= p_echec:
                succes += 1; budget.succes(); break
            if strategie == "budget" and not budget.peut_reessayer(): break
            if strategie == "aucun": break
    return appels / n, succes / n

print("échec de la dépendance | sans retry | retries illimités (3) | avec budget 1
   0 %")
for p in (0.01, 0.1, 1.0):
    a, b, c = (charge_avec(s, p) for s in ("aucun", "illimite", "budget"))
    print(f"        {p:5.0%}          | {a[0]:4.2f}× {a[1]:5.1%} |   {b[0]:4.2f}×
   {b[1]:5.1%}     |  {c[0]:4.2f}× {c[1]:5.1%}")
assert charge_avec("illimite", 1.0)[0] > 3.9 and charge_avec("budget", 1.0)[0] < 1
   .01          # panne totale : le budget arrête les relances
assert charge_avec("budget", 0.01)[1] > charge_avec("aucun", 0.01)[1]
                # panne légère : le budget masque encore les échecs
```

Exemple pas à pas (suite)

Lecture :

1. **Amplification** : 3 couches × 3 tentatives = 27 × la charge en cas de panne totale. Les retries masquent bien une panne **partielle** (30 % d'échec devient ~0 % côté utilisateur) mais **multiplient** la charge d'une dépendance déjà en difficulté.
2. **Propagation d'échéance** : avec une échéance partagée, la couche `service` constate que l'utilisateur est déjà parti et **ne démarre pas** son travail. Sans propagation, chaque couche reprend un délai neuf et travaille pour rien.
3. **Hedging** : en doublant seulement ~5 % des requêtes, le p99 chute fortement (il n'est plus déterminé par les ralentissements ponctuels), au prix de moins de 10 % de charge en plus.
4. **Budget de retries** : en panne légère, les relances passent et masquent les échecs ; en panne totale, le budget s'épuise et la charge reste proche de 1 × (au lieu de 4 ×) : on **échoue vite** sans marteler la dépendance.

À toi, avec aide

Un appel utilisateur traverse : navigateur → passerelle → service A → service B → base. Chaque appel a une latence normale et un délai (timeout) :

Appel	Latence p99	Délai choisi
passerelle → A	150 ms	2 s
A → B	100 ms	1 s
B → base	50 ms	500 ms

1. Quel est le temps maximal d'attente de l'utilisateur si **chaque** maillon réessaie 2 fois (3 tentatives) et que la base ne répond jamais ?
2. Quelle serait la bonne règle pour les délais (relation entre un délai amont et les délais aval) ? Propose des valeurs cohérentes si l'utilisateur ne doit pas attendre plus de 2 s.
3. Combien de requêtes la base reçoit-elle pour une seule requête utilisateur dans le cas 1 ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Dans le pire cas, chaque niveau attend `tentatives × délai` du niveau aval.

Indice 2

Le délai amont doit être supérieur à la somme (tentatives × délais aval), ou on propage une échéance globale.

Indice 3

La charge sur la base = produit des tentatives des couches.

Réponse

```
# 1. pire cas du point de vue de chaque niveau (la base ne répond jamais)
t_B = 3 * 0.5            # B attend la base 3 fois 500 ms
t_A = 3 * 1.0            # A attend B 3 fois 1 s (borné par son propre délai)
t_pass = 3 * 2.0         # la passerelle attend A 3 fois 2 s
print("attente de l'utilisateur dans le pire cas :", t_pass, "s")
# 3. appels à la base : passerelle(3) × A(3) × B(3)
appels_base = 3 * 3 * 3
print("appels reçus par la base :", appels_base)
assert t_pass == 6.0 and appels_base == 27
```

1. **6 secondes** pour un utilisateur qui ne devrait pas attendre plus de 2 s : les délais amont sont mal liés.
2. **Règle** : un délai amont doit être **plus grand que le total de ce que l'aval peut consommer** — ou mieux, on propage une **échéance globale** (2 s) et chaque maillon l'honore. Valeurs cohérentes : échéance 2 s ; passerelle → A : 1,8 s ; A → B : 1,2 s ; B → base : 0,4 s, avec **un seul** niveau de retry (par exemple à la passerelle, avec budget).

Réponse (suite)

3. **27 requêtes** vers la base pour 1 requête utilisateur (3 × 3 × 3) : c'est l'amplification que la mission décrit.

À toi, en solo

Client résilient complet. Écris `ClientResilient` qui combine :

1. échéance globale transmise (temps simulé) ;
2. retries avec recul exponentiel et gigue, limités par un budget (seau à jetons à 10 %) ;
3. doublage (hedging) après un seuil pour les opérations marquées idempotentes ;
4. disjoncteur (5 échecs consécutifs → ouvert pendant 2 s) ;
5. compte-rendu : nombre d'appels réels envoyés, latence perçue, issue (succès, échec, refusé par le disjoncteur).

Teste trois scénarios simulés (horloge virtuelle, graine fixée) : (a) dépendance saine, (b) 5 % d'erreurs transitoires, (c) panne totale pendant 30 s puis rétablissement. Pour chaque scénario, compare un client naïf (3 retries immédiats) et ton client : charge sur la dépendance, taux de succès, latence p99.

À toi, en solo · réponse

Pistes de correction

```
import math, random

class Dependance:
    def __init__(self, horloge, rng, p_echec=lambda t: 0.0):
        self.h, self.rng, self.p_echec, self.appels = horloge, rng, p_echec, 0
    def appeler(self):
        self.appels += 1
        base = self.rng.lognormvariate(math.log(0.02), 0.3)
        if self.rng.random() < self.p_echec(self.h["t"]):
            return False, 0.05                                  # échec rapide
        return True, base

class ClientNaif:
    def __init__(self, dep): self.dep = dep
    def executer(self, h):
        debut = h["t"]
        for _ in range(4):
            ok, d = self.dep.appeler(); h["t"] += d
            if ok: return "succes", h["t"] - debut
```

Pistes de correction (suite)

```
        return "echec", h["t"] - debut

class ClientResilient:
    def __init__(self, dep, rng, budget_ratio=0.1):
        self.dep, self.rng, self.jetons, self.ratio = dep, rng, 10.0, budget_ratio
        self.echecs_consecutifs, self.ouvert_jusqua = 0, -1
    def executer(self, h, echeance=1.0):
        debut, limite = h["t"], h["t"] + echeance
        if h["t"] < self.ouvert_jusqua: return "refuse", 0.0
        for tentative in range(4):
            if h["t"] >= limite: break                          # échéance épuisé
   e : on n'appelle plus
            ok, d = self.dep.appeler(); h["t"] += d
            if ok:
                self.jetons = min(10, self.jetons + self.ratio); self.echecs_conse
   cutifs = 0
                return "succes", h["t"] - debut
            self.echecs_consecutifs += 1
            if self.echecs_consecutifs >= 5: self.ouvert_jusqua = h["t"] + 2.0; se
```

Pistes de correction (suite)

```
    lf.echecs_consecutifs = 0; break
             if self.jetons < 1: break
             self.jetons -= 1
             h["t"] += self.rng.uniform(0, 0.05 * 2 ** tentative)  # recul exponent
    iel avec gigue
         return "echec", h["t"] - debut

def p99(v):
    v = sorted(v); return v[min(len(v) - 1, math.ceil(0.99 * len(v)) - 1)]

def scenario(nom, p_echec, client_type, n=3000, graine=5):
    rng = random.Random(graine); h = {"t": 0.0}
    dep = Dependance(h, rng, p_echec)
    client = ClientNaif(dep) if client_type == "naif" else ClientResilient(dep, rn
   g)
    resultats, latences = [], []
    for i in range(n):
        h["t"] = i * 0.02                                        # 50 requêtes par
    seconde, EN BOUCLE OUVERTE : l'arrivée ne dépend pas de la réponse
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        r, l = client.executer(h); resultats.append(r); latences.append(l)
    return dep.appels / n, resultats.count("succes") / n, p99(latences)

SCENARIOS = {"sain": lambda t: 0.0, "5 % d'erreurs": lambda t: 0.05, "panne 30 s":
    lambda t: 1.0 if 20 <= t < 50 else 0.0}
print("scénario        | client | appels/req | succès | p99")
res = {}
for nom, f in SCENARIOS.items():
    for ct in ("naif", "resilient"):
        res[(nom, ct)] = scenario(nom, f, ct)
        a, s, l = res[(nom, ct)]
        print(f"{nom:15s} | {ct:9s} | {a:5.2f} | {s:6.1%} | {l * 1000:5.0f} ms")
assert res[("panne 30 s", "resilient")][0] < res[("panne 30 s", "naif")][0] * 0.6
           # moins de charge pendant la panne
assert res[("5 % d'erreurs", "resilient")][1] > 0.97 and res[("sain", "resilient")
   ][0] < 1.01
```

À toi, en solo · réponse

Pistes de correction (suite)

Le client résilient **masque** les erreurs transitoires (comme le naïf) mais **cesse de marteler** la dépendance en panne grâce au disjoncteur et au budget — et se rétablit automatiquement ensuite. Les échéances évitent le travail inutile.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Retries à tous les niveaux	Amplification multiplicative (27 ×, 64 ×)	Un seul niveau de retry, avec budget
Délais amont plus courts que la somme des délais aval	Travail abandonné, résultats incohérents	Échéance globale propagée
Relancer immédiatement sans recul ni gigue	Tempête de retries synchronisés	Recul exponentiel + gigue
Hedging sur des opérations non idempotentes	Doublons d'effets	Seulement pour les lectures / opérations idempotentes

Mini-boss

Banc de test de résilience. Construis une simulation de la chaîne passerelle → A → B → base (horloge virtuelle, graine fixée) où tu peux configurer à chaque niveau délais, retries, disjoncteurs, hedging, et injecter : panne totale de 30 s de la base, ralentissement ponctuel (3 % de requêtes à 10×), pertes de réponses.

1. mesure pour 5 configurations (aucune protection, retries partout, retry unique avec budget, + échéance propagée, + disjoncteur et hedging) : charge sur la base, succès perçu, p50/p99, temps de rétablissement après la panne ;
2. mets en évidence au moins un cas de **panne métastable** (la charge reste élevée après la fin de la panne) ;
3. calcule l'amplification théorique et compare à la simulation ;
4. recommande une configuration, justifie chaque choix par un chiffre ;
5. écris 8 tests (propriétés : charge ≤ k × trafic, aucune requête exécutée après son échéance, pas de retry sur erreur métier).

Mini-boss · réponse

Pistes de correction

Un bon travail : une classe par mécanisme combinable, un tableau des 5 configurations (la dernière doit avoir la charge la plus faible sur la base pendant la panne et le meilleur p99), démonstration que les retries illimités maintiennent la surcharge après la fin de la panne (la file ne se vide pas) alors que le budget permet un retour à la normale en quelques secondes. L'amplification mesurée correspond à $3^3 = 27$ pour la configuration « retries partout » en panne totale.

Quiz éclair

Quiz éclair

1. Avec 3 couches qui réessaient chacune 3 fois, une dépendance en panne totale reçoit :

A. 3 fois la charge
B. 27 fois la charge normale
C. aucune charge

2. Un budget de retries sert à :

A. augmenter le nombre de relances
B. supprimer les délais d'attente
C. plafonner les relances à une fraction du trafic pour éviter l'avalanche

3. La propagation d'échéance évite que :

A. des couches travaillent pour un client qui a déjà abandonné

Quiz éclair

Quiz éclair (suite)

B. le réseau soit lent
C. les messages soient chiffrés

4. 🔁 *Révision mission 02* : Un délai dépassé sur un appel distant laisse le résultat :

A. forcément échoué
B. inconnu : l'action a pu avoir lieu ou non
C. forcément réussi

5. 🔁 *Révision mission 02* : « Exactement une fois » s'obtient en pratique par :

A. une garantie du réseau
B. un délai plus long
C. au moins une fois (relances) + idempotence

Quiz éclair · réponse

Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

Je réfléchis

1. Dans ton système, combien de couches réessaient un même appel ? Le sais-tu ?
2. Quels délais d'attente de ta chaîne d'appels sont plus longs que le délai de l'utilisateur qui attend ?

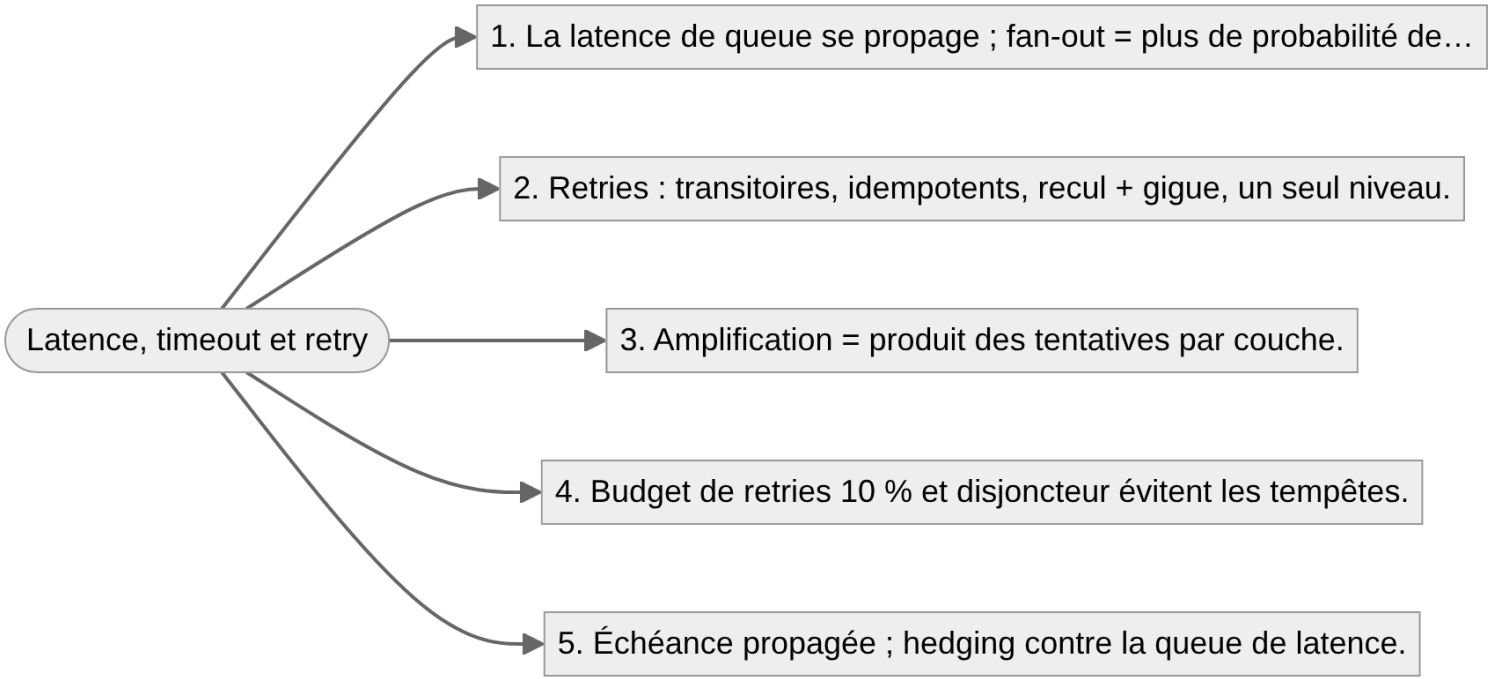
À retenir

À retenir

1. La latence de queue se propage ; fan-out = plus de probabilité de lenteur.
2. Retries : transitoires, idempotents, recul + gigue, un seul niveau.
3. Amplification = produit des tentatives par couche.
4. Budget de retries (10 %) et disjoncteur évitent les tempêtes.
5. Échéance propagée ; hedging contre la queue de latence.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Maître des délais

Prochaine mission : Idempotence

