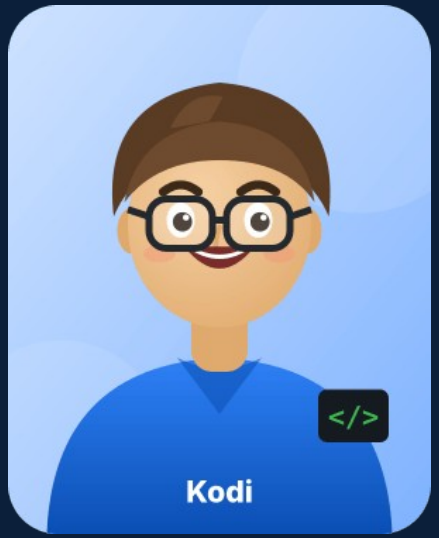
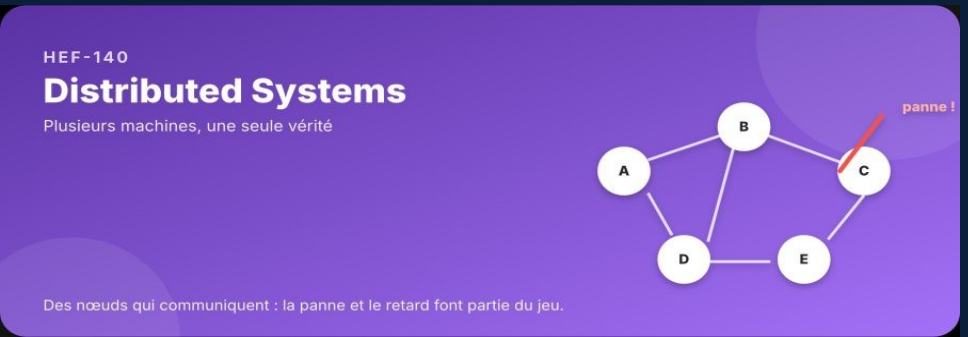
HEF-140 · MISSION 05 / 10

Réplication

Pour survivre à une panne, copie les données. Mais dès qu'il y a plusieurs copies, il faut décider laquelle croire.





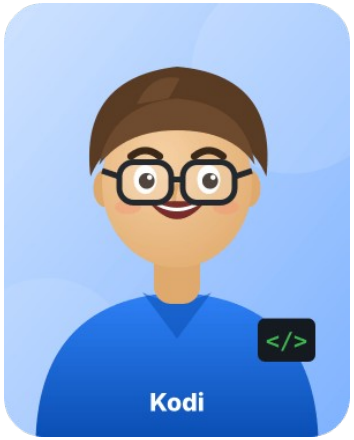
Badge : Gardien des copies

110 min · avancé

L'histoire

HashBox réplique sa base de données sur deux machines supplémentaires. Un soir, la machine principale tombe en panne ; la réplique de secours prend le relais. Le lendemain, des formateurs se plaignent : les trois dernières inscriptions de la soirée ont **disparu**. Elles avaient été confirmées à l'utilisateur, mais la réplique n'en avait pas encore reçu copie quand la principale est tombée.

« Répliquer, c'est un compromis entre vitesse, sécurité et cohérence, dit Maya. Si tu attends que toutes les copies confirment, c'est lent et fragile. Si tu n'attends pas, tu peux perdre des données. Il existe des réglages — et une règle élégante, les quorums, qui garantit qu'on lit toujours la dernière écriture. »



L'histoire

L'histoire (suite)

Tu apprends la réplication leader-suiveurs synchrone et asynchrone, le retard de réplication, le basculement, les quorums (R + W > N), et les types de données répliqués sans conflit (CRDT).

Ta mission

À la fin, je serai capable de…

1. expliquer pourquoi on réplique (disponibilité, latence, débit de lecture) et ce que cela coûte
2. comparer réplication synchrone et asynchrone (perte de données, latence)
3. simuler un basculement et mesurer les écritures perdues
4. démontrer la règle des quorums R + W > N par énumération
5. implémenter des CRDT simples (compteur, registre, ensemble) qui convergent sans coordination

Défi d'échauffement

Tu tiens un registre précieux. Tu en gardes une copie chez un ami. Soit tu lui téléphones à chaque ligne que tu écris et tu attends sa confirmation (sûr, mais lent), soit tu lui envoies les feuilles par la poste à la fin de la journée (rapide, mais il lui manque la journée en cours si ta maison brûle). Quel est ton compromis ? Que se passe-t-il si ton ami et toi corrigez la même ligne en même temps ?

Le concept

Pourquoi répliquer

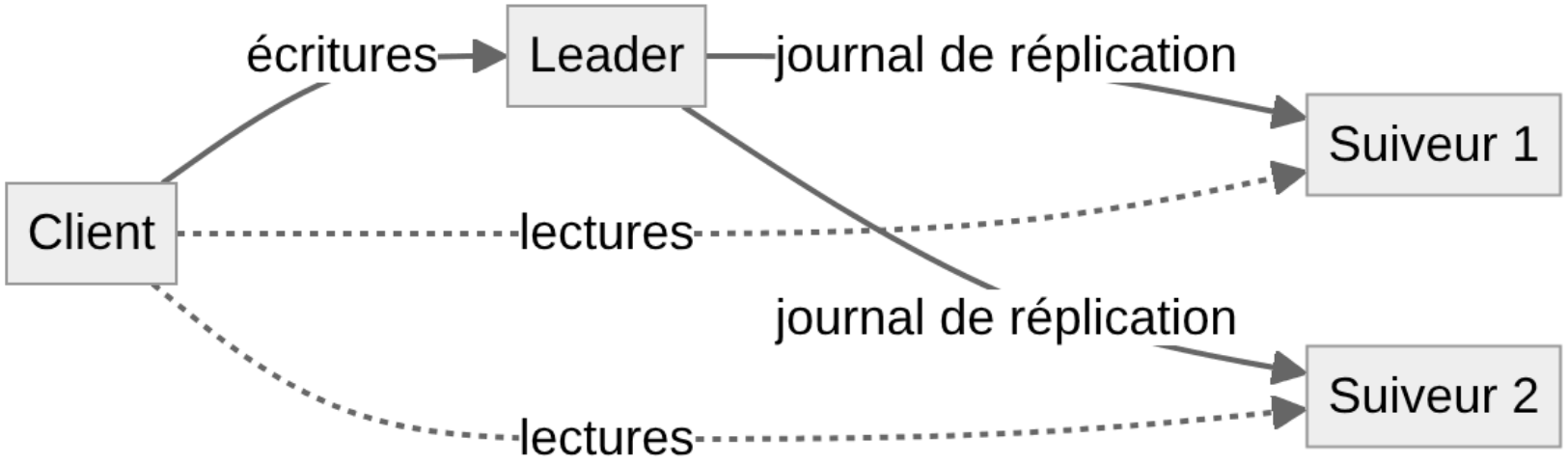
Objectif	Explication
Disponibilité	le service continue si une machine tombe
Durabilité	les données survivent à la perte d'un disque ou d'un site
Latence	des copies proches des utilisateurs
Débit de lecture	répartir les lectures sur plusieurs copies

Le coût : maintenir des copies **cohérentes** est le cœur du problème distribué.

Leader-suiveurs (primaire-réplicas)

Le concept

Leader-suiveurs (primaire-réplicas) (suite)



Description du schéma : les clients écrivent sur le leader ; le leader envoie son journal de modifications à deux suiveurs ; les lectures peuvent aller aux suiveurs.

Le concept

Leader-suiveurs (primaire-réplicas) (suite)

- Une **seule** machine accepte les écritures (pas de conflit d'écriture) ;
- les suiveurs appliquent le journal dans l'ordre.

Synchrone ou asynchrone ?

	Synchrone	Asynchrone	Semi-synchrone
Le leader confirme au client…	après que **les suiveurs ont reçu**	**immédiatement**	après **au moins un** suiveur
Latence d'écriture	élevée	faible	moyenne
Perte à la panne du leader	aucune	**possible** (écritures non répliquées)	aucune si au moins 1 suiveur survit
Disponibilité en écriture	un suiveur lent bloque	non affectée	tolère un suiveur lent

Le concept

Synchrone ou asynchrone ? (suite)

L'asynchrone est la valeur par défaut de nombreuses bases : rapide, mais l'**objectif de point de reprise (RPO)** (HEF-120) n'est pas nul.

Le retard de réplication (*replication lag*)

Avec l'asynchrone, les suiveurs sont **en retard** de quelques millisecondes à plusieurs secondes. Anomalies visibles pour l'utilisateur :

Anomalie	Exemple
Lecture périmée	je modifie mon profil, je recharge, l'ancienne version s'affiche
Violation de « lire ses écritures » (*read-your-writes*)	mon inscription n'apparaît pas

Le concept

Le retard de réplication (*replication lag*) (suite)

Anomalie	Exemple
Retour en arrière (lectures non monotones)	je vois la nouvelle valeur puis l'ancienne, selon le suiveur interrogé

Remèdes : lire sur le leader après une écriture récente ; **sessions collantes** ; attendre que le suiveur ait atteint une position donnée du journal.

Le basculement (*failover*)

Quand le leader tombe : détecter la panne (délai : fausses alertes possibles), **élire** un nouveau leader (le suiveur le plus à jour), reconfigurer les clients, **empêcher l'ancien leader de continuer à écrire** s'il revient (*split brain*, mission 9). En asynchrone : les écritures non répliquées sont **perdues** (ou en conflit au retour de l'ancien leader).

Le concept

Sans leader : les quorums

Dans un système à `N` réplicas, une écriture est considérée réussie quand `W` réplicas l'ont acceptée ; une lecture interroge `R` réplicas et prend la version la plus récente.

```
R + W > N   ⇒   toute lecture voit au moins un réplica contenant la dernière écrit
   ure
```

Les deux ensembles (écrits, lus) **se recoupent** nécessairement. Exemples avec `N = 3` : `W = 2, R = 2` (tolère 1 panne) ; `W = 3, R = 1` (écritures lentes, lectures rapides) ; `W = 1, R = 1` (rapide, **pas de garantie**).
Mécanismes complémentaires : **réparation à la lecture** (*read repair*) renvoie la bonne version aux réplicas en retard ; **anti-entropie** compare les données en arrière-plan.

Le concept

Sans leader : les quorums (suite)

⚠ *Un quorum ne garantit pas, à lui seul, la **linéarisabilité** (cas limites : écritures concurrentes, horloges, quorums « lâches »). Il garantit de voir une version récente.*

Conflits d'écriture

Sans leader unique, deux écritures concurrentes sur la même clé peuvent se produire :

Stratégie	Principe	Risque
Le dernier gagne (LWW, horloge physique)	on garde l'écriture avec l'horodatage le plus grand	**perd** silencieusement des écritures ; dépend d'horloges
Versions vectorielles (mission 1)	détecter la concurrence, garder les « frères »	le client doit fusionner
Fusion automatique (CRDT)	types de données dont la fusion est mathématiquement sûre	restreint aux types adaptés

Le concept

CRDT : types de données répliqués sans conflit

Un **CRDT** (*Conflict-free Replicated Data Type*) a une opération de **fusion** qui est :

- **commutative** : $a \sqcup b = b \sqcup a$;
- **associative** ;
- **idempotente** : $a \sqcup a = a$.

Alors, quel que soit l'ordre ou le nombre de fois où les états sont échangés, tous les réplicas **convergent** vers le même état.

CRDT	Principe
Compteur G (croissant)	un compteur par réplica ; valeur = somme ; fusion = maximum par réplica

Le concept

CRDT : types de données répliqués sans conflit (suite)

CRDT	Principe
Compteur PN	deux compteurs G (ajouts, retraits)
Registre LWW	(valeur, horodatage) ; fusion = le plus grand horodatage
Ensemble G	union
Ensemble OR	ajouts et retraits identifiés par étiquettes uniques

Utilisés pour : compteurs de « j'aime », panier, édition collaborative.

🧠 ***Astuce*** *: pour chaque donnée répliquée, décide explicitement : que faire si deux copies diffèrent ? « On verra bien » est la pire réponse.*

Exemple pas à pas

Problème : (1) simuler la réplication leader-suiveurs asynchrone, le retard, et la perte à un basculement, (2) démontrer par énumération que `R + W > N` garantit de voir la dernière écriture, (3) implémenter et vérifier par propriétés des CRDT.

```
import itertools, random

# ---------------- 1. réplication asynchrone et basculement ----------------
class Replica:
    def __init__(self, nom): self.nom, self.journal, self.applique = nom, [], 0
    def valeur(self, cle):
        for k, v in reversed(self.journal):
            if k == cle: return v

class Cluster:
    def __init__(self, mode):
        self.leader, self.suiveurs, self.mode = Replica("L"), [Replica("F1"), Repl
   ica("F2")], mode
        self.en_vol = []                                          # entrées env
```

Exemple pas à pas (suite)

```
oyées mais pas encore reçues par les suiveurs
 def ecrire(self, cle, valeur):
     self.leader.journal.append((cle, valeur))
     entree = (cle, valeur)
     if self.mode == "synchrone":
         for f in self.suiveurs: f.journal.append(entree)         # on attend l
es suiveurs avant de confirmer
     else:
         self.en_vol.append(entree)                               # confirmatio
n immédiate ; réplication plus tard
     return "confirmé"
 def repliquer(self, n=None):                                     # livre `n` e
ntrées en vol aux suiveurs
     for e in self.en_vol[:n]:
         for f in self.suiveurs: f.journal.append(e)
     self.en_vol = self.en_vol[n:] if n is not None else []
 def basculer(self):                                              # le leader t
ombe : le suiveur 1 devient leader
     nouveau = self.suiveurs[0]
```

Exemple pas à pas (suite)

```
        perdues = [e for e in self.leader.journal if e not in nouveau.journal]
        return nouveau, perdues

asyn = Cluster("asynchrone")
for i in range(10): asyn.ecrire(f"inscription:{i}", "ok")
asyn.repliquer(7)                                              # 7 entrées o
   nt eu le temps d'arriver, 3 sont encore en vol
print("lecture sur un suiveur d'une écriture confirmée récente :", asyn.suiveurs[0
   ].valeur("inscription:9"), "(retard de réplication)")
nouveau, perdues = asyn.basculer()
print(f"basculement asynchrone : {len(perdues)} écritures confirmées au client son
   t PERDUES : {[k for k, _ in perdues]}")
assert asyn.suiveurs[0].valeur("inscription:9") is None and len(perdues) == 3

sync = Cluster("synchrone")
for i in range(10): sync.ecrire(f"inscription:{i}", "ok")
_, perdues = sync.basculer()
assert perdues == []                                           # synchrone :
    aucune perte (au prix de la latence)
```

Exemple pas à pas (suite)

```
# ---------------- 2. quorums : on énumère TOUS les cas ----------------
def quorum_garantit_lecture_fraiche(N, W, R):
    reps = range(N)
    for ecrits in itertools.combinations(reps, W):                    # n'importe
   quels W réplicas ont reçu la dernière écriture
        for lus in itertools.combinations(reps, R):                   # n'importe
   quels R réplicas sont interrogés
            if not set(ecrits) & set(lus): return False               # un cas où
   la lecture ne voit AUCUN réplica à jour
    return True

print("N W R | R+W>N | garantie vérifiée par énumération")
for N, W, R in [(3, 2, 2), (3, 3, 1), (3, 1, 3), (3, 1, 1), (3, 2, 1), (5, 3, 3),
   (5, 2, 3), (5, 4, 2)]:
    garantie = quorum_garantit_lecture_fraiche(N, W, R)
    print(f"{N} {W} {R} |  {R + W > N!s:5} | {garantie}")
    assert garantie == (R + W > N)                                    # la règle e
   st exacte
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# ---------------- 3. CRDT : compteur G, registre LWW, ensemble OR ---------------
   -
class CompteurG:
    def __init__(self, replica): self.replica, self.c = replica, {}
    def incrementer(self, n=1): self.c[self.replica] = self.c.get(self.replica, 0)
    + n
    def valeur(self): return sum(self.c.values())
    def fusionner(self, autre):
        for r, v in autre.c.items(): self.c[r] = max(self.c.get(r, 0), v)
     # maximum par réplica

class RegistreLWW:
    def __init__(self): self.valeur, self.t, self.replica = None, (0, ""), ""
    def ecrire(self, valeur, t, replica): self.fusionner_valeur(valeur, (t, replic
   a))
    def fusionner_valeur(self, valeur, tag):
        if tag > self.t: self.valeur, self.t = valeur, tag
     # (horodatage, nom du réplica) : départage les égalités
```

Exemple pas à pas (suite)

```
    def fusionner(self, autre): self.fusionner_valeur(autre.valeur, autre.t)

# propriété : quel que soit l'ordre des fusions, tous les réplicas convergent
rng = random.Random(1)
for essai in range(200):
    reps = [CompteurG(f"r{i}") for i in range(4)]
    for r in reps:
        for _ in range(rng.randint(0, 5)): r.incrementer(rng.randint(1, 3))
    attendu = sum(v for r in reps for v in r.c.values())
    for _ in range(30):                                                  # fu
   sions aléatoires, avec répétitions
        a, b = rng.sample(reps, 2); a.fusionner(b)
    for a in reps:                                                       # un
   e dernière passe complète garantit la convergence
        for b in reps: a.fusionner(b)
    assert all(r.valeur() == attendu for r in reps)
print("compteur G : 200 scénarios aléatoires, tous les réplicas convergent vers la
    somme exacte")
```

Exemple pas à pas (suite)

```
a, b = RegistreLWW(), RegistreLWW()
a.ecrire("prix=45", 100, "paris"); b.ecrire("prix=50", 100, "lyon")             #
   écritures concurrentes à la même seconde
a1, b1 = RegistreLWW(), RegistreLWW()
a1.fusionner(a); a1.fusionner(b); b1.fusionner(b); b1.fusionner(a)              #
   ordres de fusion différents
assert a1.valeur == b1.valeur == "prix=45"                                      #
    même résultat (départage déterministe : « paris » > « lyon »)
print("registre LWW : une des deux écritures concurrentes est perdue silencieuseme
   nt ; le gagnant :", a1.valeur)
```

Lecture :

1. **Asynchrone** : le client a reçu 10 confirmations, mais le suiveur n'avait que 7 écritures quand le leader est tombé : **3 écritures confirmées sont perdues** après basculement. En **synchrone**, aucune perte — au prix d'une latence d'écriture plus élevée et d'une disponibilité plus fragile.

Exemple pas à pas (suite)

2. **Quorums** : l'énumération de **tous** les choix de réplicas écrits et lus montre que la garantie « la lecture voit un réplica à jour » existe **si et seulement si** `R + W > N`. Les configurations `(3,1,1)`, `(3,2,1)`, `(5,2,3)` n'ont pas la garantie.
3. **CRDT** : après des fusions dans un ordre quelconque (avec répétitions), les compteurs convergent vers la somme exacte. Le registre **LWW** converge aussi, mais **perd** l'une des deux écritures concurrentes : convergent n'est pas synonyme de « sans perte ».

À toi, avec aide

Un système réplique chaque donnée sur **N = 5** réplicas, chacun disponible 99 % du temps (indépendamment).

1. Pour `W = 3, R = 3` (quorum majoritaire) : quelle est la probabilité qu'une écriture puisse être effectuée (au moins W réplicas disponibles) ? Même question pour `W = 5` (tous).
2. Quel est le nombre maximal de réplicas en panne que `W = 3, R = 3` tolère ?
3. Pour `W = 1` et `R = 5`, quel est le risque (pour la cohérence) ? Que gagne-t-on ?

Indices progressifs

Indice 1

P(au moins k disponibles) = somme de la loi binomiale pour i ≥ k.

Indice 2

`math.comb(5, i) * 0.99**i * 0.01**(5-i)`.

Indice 3

Avec R + W = 6 > 5, la garantie existe ; elle n'existe plus si W = 1 et R = 3.

Réponse

```
from math import comb
def p_au_moins(k, n=5, p=0.99):
    return sum(comb(n, i) * p**i * (1 - p)**(n - i) for i in range(k, n + 1))
print(f"W = 3 : disponibilité en écriture {p_au_moins(3):.8f}")
print(f"W = 5 : disponibilité en écriture {p_au_moins(5):.6f}")
assert p_au_moins(3) > 0.99999 and abs(p_au_moins(5) - 0.99 ** 5) < 1e-12
```

1. **W = 3** : probabilité ≈ 0,99999 (cinq neuf après la virgule : environ 5 minutes d'indisponibilité par an) ; **W = 5** : ≈ 0,951 (seulement 95 %, soit plus de 18 jours d'indisponibilité par an !). Exiger **tous** les réplicas rend le système **moins** disponible qu'une seule machine.
2. `W = 3, R = 3` tolère **2 réplicas en panne** (il en reste 3 pour constituer un quorum).
3. `W = 1, R = 5` : `R + W = 6 > 5`, donc lire les 5 réplicas voit toujours la dernière écriture — mais **la lecture exige que les 5 soient disponibles** (une panne la bloque). On gagne des **écritures très rapides et disponibles** (1 réplica suffit), on paie des **lectures lentes et fragiles**. Si l'on baisse à `R = 3`, on perd la garantie (`3 + 1 < 5`).

À toi, en solo

Magasin à quorum avec réparation à la lecture. Implémente un magasin clé-valeur à 5 réplicas :

1. chaque valeur porte une **version** (compteur) ; `ecrire(cle, valeur, W)` l'envoie à W réplicas **disponibles** choisis au hasard (graine fixée) ; si moins de W sont disponibles, l'écriture échoue sans rien appliquer partiellement de façon visible (ou documente le comportement) ;
2. `lire(cle, R)` interroge R réplicas disponibles, renvoie la valeur de version maximale, et **répare** les réplicas lus qui sont en retard (*read repair*) ;
3. simule 1 000 opérations aléatoires avec pannes aléatoires de réplicas ; avec `R + W > N`, vérifie qu'**aucune lecture réussie ne renvoie une version plus ancienne que la dernière écriture réussie** ; avec `W = R = 1`, mesure le taux de lectures périmées ;
4. mesure le taux d'échecs (indisponibilité) pour `W = R = 3` contre `W = R = 5` avec 5 % de pannes par réplica.

À toi, en solo · réponse

Pistes de correction

```
import random

class Magasin:
    def __init__(self, n=5, graine=1):
        self.n, self.rng = n, random.Random(graine)
        self.reps = [{} for _ in range(n)]                          # cle -> (version
   , valeur)
        self.en_panne = set(); self.derniere = {}                   # dernière écritu
   re RÉUSSIE par clé (pour la vérification)
    def disponibles(self): return [i for i in range(self.n) if i not in self.en_pa
   nne]
    def ecrire(self, cle, valeur, W):
        dispo = self.disponibles()
        if len(dispo) < W: return False
        version = self.derniere.get(cle, 0) + 1
        for i in self.rng.sample(dispo, W):
            self.reps[i][cle] = (version, valeur)
        self.derniere[cle] = version; return True
    def lire(self, cle, R):
```

Pistes de correction (suite)

```
        dispo = self.disponibles()
        if len(dispo) < R: return (0, None), False
        choisis = self.rng.sample(dispo, R)
        meilleur = max((self.reps[i].get(cle, (0, None)) for i in choisis), key=la
    mbda x: x[0])
        for i in choisis:                                    # réparation à
    la lecture
            if self.reps[i].get(cle, (0, None))[0] < meilleur[0]: self.reps[i][cle
    ] = meilleur
        return meilleur, True

def simuler(W, R, p_panne=0.05, n_ops=3000, graine=2):
    rng, m = random.Random(graine), Magasin(graine=graine)
    perimees = lectures = echecs = 0
    for op in range(n_ops):
        m.en_panne = {i for i in range(5) if rng.random() < p_panne}
        cle = f"k{rng.randrange(5)}"
        if rng.random() < 0.5:
            if not m.ecrire(cle, op, W): echecs += 1
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        else:
            (version, _), ok = m.lire(cle, R)
            if not ok: echecs += 1; continue
            lectures += 1
            if version < m.derniere.get(cle, 0): perimees += 1
    return perimees / max(lectures, 1), echecs / n_ops

for W, R in [(3, 3), (5, 1), (1, 5), (1, 1), (2, 2)]:
    perimees, echecs = simuler(W, R)
    print(f"W={W} R={R} (R+W>N : {R + W > 5!s:5}) → lectures périmées {perimees:6.
   2%} | opérations échouées {echecs:6.2%}")
    if R + W > 5: assert perimees == 0.0
assert simuler(1, 1)[0] > 0.05
assert simuler(5, 5)[1] > 3 * simuler(3, 3)[1]                    # exiger tous
    les réplicas est bien plus fragile
```

À toi, en solo · réponse

Pistes de correction (suite)

La règle `R + W > N` donne **zéro lecture périmée** dans la simulation ; avec `W = R = 1`, une grande partie des lectures (de l'ordre de 80 %) sont périmées. Mais exiger **tous** les réplicas (`W = R = 5`) multiplie les échecs : plus la garantie est forte, plus la disponibilité est fragile. La réparation à la lecture **soigne** progressivement les retards.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Confirmer au client avant la réplication, sans le dire	Écritures confirmées puis perdues après basculement	Choisir et documenter synchrone / semi-synchrone / asynchrone
Lire sur un suiveur juste après une écriture	L'utilisateur ne voit pas sa propre modification	Lire sur le leader ou attendre la position du journal
« Le dernier gagne » par horloge physique	Perte silencieuse d'écritures	Versions vectorielles, CRDT ou arbitrage explicite
Quorum mal calibré (R + W ≤ N)	Lectures périmées sans le savoir	Vérifier R + W > N ; tester par énumération

Mini-boss

Base répliquée de HashBox. Construis une simulation d'une base clé-valeur répliquée avec trois modes (leader asynchrone, leader semi-synchrone, quorum 3/5) et un injecteur de pannes (arrêt du leader, partition, retard de réplication).

1. implémente les trois modes avec le journal de réplication, le retard, le basculement ;
2. mesure pour chaque mode, sur 200 scénarios aléatoires : écritures confirmées perdues, lectures périmées, lectures non monotones, disponibilité en écriture et en lecture, latence d'écriture simulée ;
3. ajoute la garantie « lire ses écritures » (session collante) et mesure son effet sur le débit de lecture du leader ;
4. ajoute un compteur CRDT pour le nombre d'inscriptions et un registre LWW pour le prix d'un cours, et montre sur un scénario de partition ce que chacun perd ou conserve ;
5. rédige un tableau de décision : pour chaque type de donnée de HashBox (inscriptions, prix, compteur de vues, profil), le mode de réplication recommandé et pourquoi.

Mini-boss · réponse

Pistes de correction

Un bon travail : les trois modes mesurés sur les mêmes scénarios aléatoires; le semi-synchrone n'a aucune perte pour une panne simple du leader; l'asynchrone perd des écritures proportionnellement au retard; le quorum 3/5 n'a pas de lecture périmée avec R+W>N; tableau de décision cohérent (inscriptions et paiements : synchrone ou quorum ; compteurs de vues : CRDT; prix : leader unique ; profil : asynchrone avec lecture du leader après écriture).

Quiz éclair

Quiz éclair

1. En réplication asynchrone, quand le leader tombe :

A. aucune écriture n'est jamais perdue
B. les suiveurs sont toujours à jour
C. des écritures déjà confirmées au client peuvent être perdues

2. La règle de quorum qui garantit de lire au moins un réplica à jour est :

A. R + W > N
B. R + W < N
C. R = W = 1

3. Un CRDT converge parce que sa fusion est :

A. aléatoire

Quiz éclair

Quiz éclair (suite)

B. commutative, associative et idempotente
C. irréversible

4. 🔁 *Révision mission 04* : Une clé d'idempotence doit être :

A. nouvelle à chaque tentative
B. secrète et chiffrée
C. la même pour toutes les relances d'une même intention

5. 🔁 *Révision mission 04* : Pour qu'un consommateur de messages soit idempotent, on enregistre l'identifiant du message :

A. dans la même transaction que l'effet
B. dans un journal séparé, plus tard

Quiz éclair

Quiz éclair (suite)

C. jamais

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Si la machine principale de ta base tombait maintenant, combien d'écritures confirmées pourrais-tu perdre ?
2. Pour quelle donnée de ton système accepterais-tu une valeur légèrement périmée, et pour laquelle jamais ?

À retenir

À retenir

1. Réplication leader-suiveurs : synchrone (sûr, lent), asynchrone (rapide, perte possible), semi-synchrone.
2. Retard de réplication : lectures périmées, lecture de ses écritures, retours en arrière.
3. Basculement : détection, élection, empêcher l'ancien leader d'écrire.
4. Quorum : R + W > N garantit de voir au moins un réplica à jour.
5. CRDT : fusion commutative, associative, idempotente ; LWW perd des écritures.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Gardien des copies

Prochaine mission : Partitionnement

