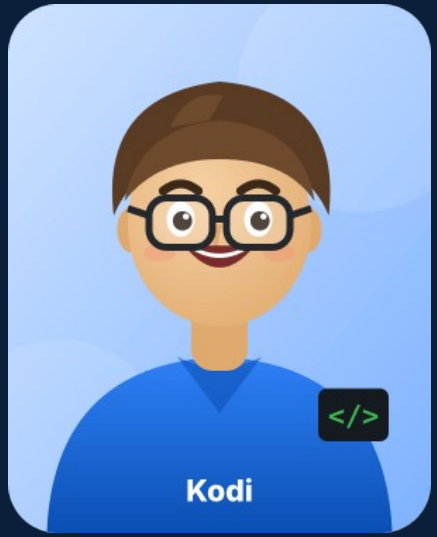
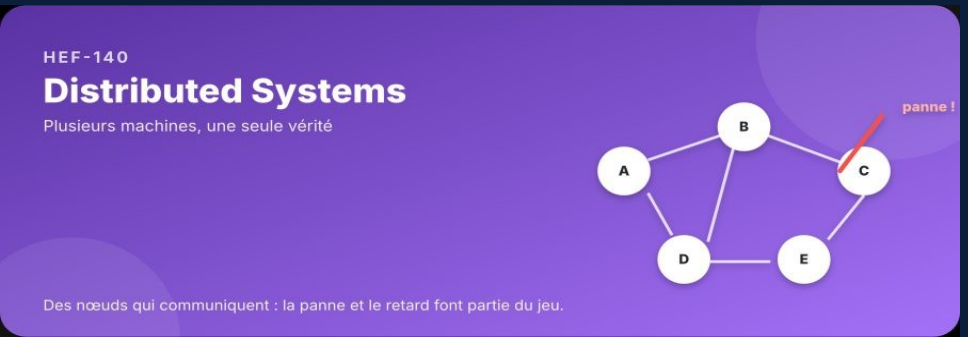
HEF-140 · MISSION 09 / 10

Tolérance aux pannes

On ne supprime pas les pannes ; on conçoit pour qu'elles ne deviennent pas des catastrophes. Détecter, isoler, basculer — sans se tromper de coupable.



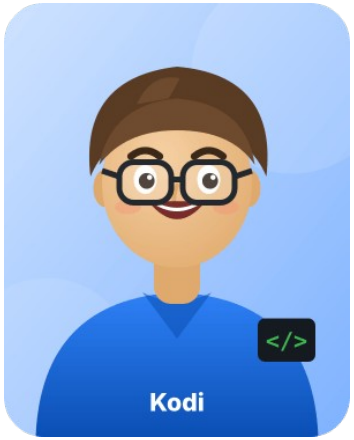


Badge : Bâtisseur de résilience

110 min · avancé

L'histoire

Un jeudi, le serveur de verrous de HashBox est lent pendant quelques secondes. Le client A, qui détient le verrou d'édition d'un cours, se met en pause (ramasse-miettes). Son bail expire, le client B obtient le verrou et écrit. A se réveille, croit toujours détenir le verrou, et écrit **par-dessus**. Le cours contient un mélange des deux modifications.



« Les deux clients ont fait ce qu'il fallait, dit Sam. Le défaut est dans l'hypothèse : "si je pense avoir le verrou, je l'ai". En distribué, on peut croire détenir un droit qu'on a perdu. La parade s'appelle un jeton de barrière : le stockage lui-même refuse les écritures d'un détenteur périmé. »

L'histoire (suite)

Tu apprends à détecter les pannes (battements de cœur, faux positifs), à calculer la disponibilité d'architectures, à utiliser baux et jetons de barrière contre les « cerveaux fendus », et à gérer les transactions distribuées : validation en deux phases (et son blocage) contre sagas avec compensations.

Ta mission

À la fin, je serai capable de…

1. calculer la disponibilité de composants en série et en parallèle
2. expliquer le compromis du détecteur de pannes (rapidité de détection contre faux positifs)
3. utiliser baux et jetons de barrière pour empêcher les écritures d'un ancien détenteur
4. comparer validation en deux phases (2PC) et saga avec compensations
5. concevoir des étapes compensables et idempotentes

Défi d'échauffement

Défi d'échauffement

Un hôpital a deux générateurs de secours et une alimentation du réseau. L'alimentation seule est disponible 99,9 % du temps ; chaque générateur 95 %. Si l'hôpital tombe quand tout le courant est coupé, quelle est la disponibilité globale ? Que se passerait-il si le basculement automatique (qui décide de passer sur les générateurs) tombait en panne lui-même ?

Le concept

Disponibilité

La **disponibilité** est la fraction du temps où le service fonctionne. On parle en « neuf » :

Disponibilité	Indisponibilité annuelle
99 %	3,65 jours
99,9 %	8,76 heures
99,99 %	52,6 minutes
99,999 %	5,26 minutes

Composition (pannes indépendantes) :

- **en série** (tous nécessaires) : $A = A_1 \times A_2 \times \dots$ — la disponibilité **baisse** ;

Le concept

Disponibilité (suite)

- **en parallèle** (redondance, un seul suffit) : $A = 1 - (1 - A_1)(1 - A_2)\dots$ — elle **monte**.

Dix services à 99,9 % en série : $0{,}999^{10} \approx 99{,}0\ \%$. Redonder la partie la plus fragile est le gain le plus rentable. Mais **les pannes ne sont pas indépendantes** (même alimentation, même version déployée, même configuration) : le calcul est une borne optimiste.

Détecter les pannes

On ne peut pas distinguer « mort » de « lent » (mission 1). Un **détecteur de pannes** suspecte un nœud quand il n'a pas reçu de **battement de cœur** (*heartbeat*) depuis un délai T.

T court	T long
détection rapide	détection lente
faux positifs : un nœud lent est déclaré mort	peu de faux positifs

Le concept

Détecter les pannes (suite)

Un faux positif est coûteux : basculement inutile, deux leaders (« cerveau fendu » *split brain*), charge de rééquilibrage. Raffinements : détecteurs **adaptatifs** (phi-accrual : seuil fondé sur la distribution observée des délais), **suspicion** par plusieurs nœuds, confirmation avant action.

Cerveau fendu, baux et jetons de barrière

Un **bail** (*lease*) est un droit **temporaire** (un verrou avec expiration) accordé par un service de coordination. Problème : un client en pause peut **croire** détenir encore son bail alors qu'il a expiré et qu'un autre l'a obtenu.

Jeton de barrière (*fencing token*) :

1. à chaque octroi de bail, le service délivre un **numéro croissant** (1, 2, 3…) ;
2. le client joint son numéro à chaque écriture ;

Le concept

Cerveau fendu, baux et jetons de barrière (suite)

3. la ressource protégée **mémorise le plus grand numéro vu** et **refuse** toute écriture portant un numéro plus petit.

Ainsi un ancien détenteur ne peut plus rien écrire, même s'il se croit légitime. La **sûreté** est garantie par la ressource, pas par la ponctualité des clients.

Transactions distribuées

Une opération qui touche plusieurs services (réserver une place, débiter, notifier) doit être **tout ou rien**.

Validation en deux phases (2PC)

1. **Préparation** : le coordinateur demande à tous les participants « pouvez-vous valider ? » ; chacun verrouille les ressources, écrit « prêt » durablement, répond oui/non.

Le concept

Validation en deux phases (2PC) (suite)

2. **Validation ou annulation** : si tous ont dit oui, le coordinateur ordonne `commit`, sinon `abort`.

Défaut majeur : le blocage. Si le coordinateur tombe **après** la préparation, les participants « prêts » ne peuvent ni valider ni annuler seuls (peut-être les autres ont-ils déjà validé ?) : ils **gardent leurs verrous** jusqu'au retour du coordinateur. Peu adapté aux services indépendants et aux longues latences.

Sagas

Une **saga** découpe l'opération en **étapes locales**, chacune avec une **compensation** (action inverse) :

```
réserver place → débiter → envoyer confirmation
      ↑ annuler place   ↑ rembourser
```

Si une étape échoue, on exécute **les compensations dans l'ordre inverse** des étapes déjà faites. Pas de verrou global, mais :

Le concept

Sagas (suite)

- les états intermédiaires sont **visibles** (isolation faible) ;
- les compensations doivent être **idempotentes** et **réessayées** jusqu'au succès ;
- certaines actions ne sont **pas compensables** (e-mail envoyé) : les placer **en dernier** (étape pivot).

Deux styles : **chorégraphie** (chaque service réagit aux événements des autres) et **orchestration** (un orchestrateur pilote les étapes).

Dégradation progressive

Quand une dépendance tombe : continuer avec une fonction réduite (afficher sans les recommandations, servir un cache, file d'attente locale), plutôt que tout arrêter. À décider **à la conception**, par fonction.

Le concept

Principes de conception

Principe	Application
Éviter les **points de défaillance uniques**	redondance, y compris pour le basculement lui-même
Isoler	cloisons, une panne ne se propage pas
Échouer vite	délais courts, disjoncteurs
Réparer vite	MTTR faible, automatisation (HEF-120)
Tester la panne	ingénierie du chaos

🧠 ***Astuce*** *: pour chaque composant, écris : « s'il tombe, qui s'en aperçoit, en combien de temps, et qu'est-ce qui continue de marcher ? » Si personne ne s'en aperçoit, c'est le premier problème.*

Exemple pas à pas

Problème : (1) calculer des disponibilités, (2) mesurer le compromis du détecteur de pannes, (3) montrer qu'un jeton de barrière empêche l'écriture d'un ancien détenteur, (4) illustrer le blocage de la validation en deux phases et (5) exécuter une saga avec compensations, y compris quand une compensation doit être réessayée.

```
import math, random

# ---------------- 1. disponibilité ----------------
def serie(*a): return math.prod(a)
def parallele(*a): return 1 - math.prod(1 - x for x in a)
print(f"10 services à 99,9 % en série : {serie(*[0.999] * 10):.4%}")
print(f"2 instances à 99 % en parallèle : {parallele(0.99, 0.99):.4%} ; 3 instance
   s : {parallele(*[0.99] * 3):.6%}")
chaine = serie(0.9999, parallele(0.99, 0.99), 0.999)          # équilibreur × (2 s
   erveurs en parallèle) × base
print(f"équilibreur 99,99 % × 2 serveurs à 99 % × base 99,9 % = {chaine:.4%}  → {(
   1 - chaine) * 525600:.0f} min d'indisponibilité/an")
```

Exemple pas à pas (suite)

```
assert round(serie(*[0.999] * 10), 3) == 0.990 and round(parallele(0.99, 0.99), 4)
    == 0.9999

# ---------------- 2. détecteur de pannes : rapidité contre faux positifs --------
   --------
def tester_detecteur(T, graine=1):
    rng = random.Random(graine)
    # battements toutes les 1 s, avec délai réseau variable (lognormal, queue lour
   de) ; le nœud tombe à t = 3600 s
    t, dernier, faux, detection = 0.0, 0.0, 0, None
    arrivees = []
    for s in range(7200):
        if s >= 3600: break
        arrivees.append(s + rng.lognormvariate(math.log(0.05), 0.9))        # heur
   e d'arrivée du battement envoyé à la seconde s
    arrivees.sort(); dernier = 0.0
    suspect = False
    for s in range(1, 3601):
        recus = [a for a in arrivees if dernier < a <= s]
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        if recus: dernier = max(recus)
        if s - dernier > T and not suspect: faux += 1; suspect = True
     # suspicion sur un nœud VIVANT
        if s - dernier <= T: suspect = False
    # puis il tombe : la dernière arrivée est ~3600 ; on détecte quand l'écart dép
   asse T
    detection = T
     # le temps de détection est de l'ordre de T
    return faux, detection
print("délai T | faux positifs sur 1 h | temps de détection ≈")
res = {}
for T in (0.5, 1, 2, 5):
    res[T] = tester_detecteur(T); print(f"  {T:4.1f} s |        {res[T][0]:4d}
         |  {res[T][1]:.1f} s")
assert res[0.5][0] > res[2][0] >= res[5][0] and res[5][1] > res[0.5][1]

# ---------------- 3. bail + jeton de barrière ----------------
class ServiceBail:
    def __init__(self): self.numero, self.detenteur, self.expire = 0, None, 0
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    def acquerir(self, client, maintenant, duree=10):
        if self.detenteur is None or maintenant >= self.expire:
            self.numero += 1; self.detenteur, self.expire = client, maintenant + d
   uree
            return self.numero                                  # jeton croiss
   ant
        return None
class Stockage:
    def __init__(self, avec_barriere): self.avec_barriere, self.max_jeton, self.co
   ntenu, self.journal = avec_barriere, 0, "", []
    def ecrire(self, client, jeton, texte):
        if self.avec_barriere and jeton < self.max_jeton:
            self.journal.append(f"{client} (jeton {jeton}) REFUSÉ"); return False
        self.max_jeton = max(self.max_jeton, jeton); self.contenu += f"[{client}:{
   texte}]"; self.journal.append(f"{client} (jeton {jeton}) accepté"); return True

for avec in (False, True):
    bail, st = ServiceBail(), Stockage(avec)
    jeton_a = bail.acquerir("A", maintenant=0)                   # A obtient l
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    e bail (jeton 1)
     # A se met en pause (ramasse-miettes, longue) ; le bail expire à t = 10
     jeton_b = bail.acquerir("B", maintenant=12)                    # B obtient l
    e bail (jeton 2)
     st.ecrire("B", jeton_b, "version B")
     st.ecrire("A", jeton_a, "version A périmée")                   # A se réveil
    le : il croit toujours détenir le bail
     print(f"{'avec' if avec else 'sans'} jeton de barrière : contenu = {st.contenu
    }  | {st.journal}")
     assert (("version A périmée" not in st.contenu) == avec)

# ---------------- 4. 2PC : le blocage après la panne du coordinateur ------------
    ----
class Participant:
    def __init__(self, nom, vote=True): self.nom, self.vote, self.etat = nom, vote
    , "initial"
    def preparer(self):
        if self.vote: self.etat = "prêt (verrous tenus)"; return True
        self.etat = "annulé"; return False
```

Exemple pas à pas (suite)

```
    def valider(self): self.etat = "validé"
    def annuler(self): self.etat = "annulé"

def deux_phases(participants, planter_apres_preparation=False):
    votes = [p.preparer() for p in participants]
    if planter_apres_preparation: raise RuntimeError("le coordinateur tombe après
   la phase de préparation")
    (lambda: [p.valider() for p in participants])() if all(votes) else [p.annuler(
   ) for p in participants]

ps = [Participant("places"), Participant("paiement"), Participant("notif")]
try: deux_phases(ps, planter_apres_preparation=True)
except RuntimeError as e: print(e)
print("états des participants :", {p.nom: p.etat for p in ps})
assert all(p.etat == "prêt (verrous tenus)" for p in ps)          # BLOQUÉS : ni
   valider ni annuler seuls

# ---------------- 5. saga avec compensations ----------------
class Saga:
```

Exemple pas à pas

Exemple pas à pas (suite)

```
 def __init__(self, etapes): self.etapes = etapes                  # liste de (
nom, action, compensation)
 def executer(self, contexte, echoue_a=None, compensation_instable=None):
     faites = []
     for nom, action, compensation in self.etapes:
         try:
             if nom == echoue_a: raise RuntimeError(f"étape {nom} en échec")
             action(contexte); faites.append((nom, compensation))
         except RuntimeError as e:
             for nom_c, comp in reversed(faites):                    # compensa
tions dans l'ordre INVERSE
                 for essai in range(5):                              # idempote
ntes, réessayées jusqu'au succès
                     try:
                         if nom_c == compensation_instable and essai < 2: raise
 ConnectionError("compensation momentanément indisponible")
                         comp(contexte); break
                     except ConnectionError: continue
                 else: raise AssertionError(f"compensation {nom_c} impossible :
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    intervention humaine")
                return f"saga annulée ({e})"
        return "saga terminée"

def ctx(): return {"places": 5, "solde": 100, "journal": [], "rembourse": False}
def reserver(c): c["places"] -= 1; c["journal"].append("place réservée")
def liberer(c):
    if "place libérée" not in c["journal"]: c["places"] += 1; c["journal"].append(
   "place libérée")           # idempotente
def debiter(c): c["solde"] -= 40; c["journal"].append("débité")
def rembourser(c):
    if not c["rembourse"]: c["solde"] += 40; c["rembourse"] = True; c["journal"].a
   ppend("remboursé")        # idempotente
def confirmer(c): c["journal"].append("confirmation envoyée")
SAGA = Saga([("place", reserver, liberer), ("paiement", debiter, rembourser), ("co
   nfirmation", confirmer, lambda c: None)])

c1 = ctx(); assert SAGA.executer(c1) == "saga terminée" and c1["places"] == 4 and
   c1["solde"] == 60
```

Exemple pas à pas (suite)

```
c2 = ctx(); r = SAGA.executer(c2, echoue_a="confirmation", compensation_instable="
   paiement")
print(r, "→", c2["journal"], "| places", c2["places"], "| solde", c2["solde"])
assert c2["places"] == 5 and c2["solde"] == 100 and c2["journal"][-2:] == ["rembou
   rsé", "place libérée"]       # compensations dans l'ordre inverse
```

Lecture :

1. **Série/parallèle** : dix services à 99,9 % en série donnent ≈ 99 % ; doubler un composant à 99 % passe à 99,99 %. Redonder le maillon faible est le gain le plus rentable.
2. **Détecteur** : un délai court détecte vite mais suspecte souvent un nœud vivant (faux positifs) ; un délai long fait l'inverse. Il n'existe pas de bonne valeur universelle : on arbitre.
3. **Jeton de barrière** : sans jeton, l'écriture périmée de A se mélange à celle de B ; avec jeton, **le stockage refuse** l'écriture portant le jeton 1 après avoir vu le jeton 2.

Exemple pas à pas (suite)

4. **2PC** : le coordinateur tombe après la préparation et les trois participants restent **bloqués**, verrous tenus, incapables de décider seuls.
5. **Saga** : l'étape de confirmation échoue ; les compensations s'exécutent dans l'ordre inverse (remboursement, puis libération de la place), la compensation du paiement est **réessayée** deux fois avant de réussir, et elle est **idempotente**.

À toi, avec aide

Une application a l'architecture suivante : un équilibreur (99,99 %), deux serveurs d'application en parallèle (99,5 % chacun), un service d'authentification externe (99,9 %), et une base avec un leader et un réplica (chacun 99,9 %, basculement automatique fiable à 99 %).

1. Calcule la disponibilité globale (modélise le groupe base : disponible si au moins un des deux est disponible **et** le basculement fonctionne, ou le leader seul est disponible) ;
2. quel est le maillon qui limite le plus la disponibilité globale ? ;
3. combien de minutes d'indisponibilité par an ?
4. quelle amélioration apporte le plus (ajouter un 3e serveur d'application, doubler le service d'authentification, ou améliorer le basculement à 99,9 %) ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Groupe base ≈ `A_leader + (1 − A_leader) × A_replica × A_basculement`.

Indice 2

Les serveurs d'application en parallèle : `1 − (1 − 0,995)²`.

Indice 3

Calcule la variation de disponibilité globale pour chaque amélioration.

À toi, avec aide · réponse

Réponse

```
import math
def serie(*a): return math.prod(a)
def parallele(*a): return 1 - math.prod(1 - x for x in a)

def systeme(n_app=2, auth=0.999, bascule=0.99):
    base = 0.999 + (1 - 0.999) * 0.999 * bascule               # leader OK, ou (le
   ader KO et réplica OK et basculement OK)
    return serie(0.9999, parallele(*[0.995] * n_app), auth, base)

ref = systeme()
print(f"globale : {ref:.5%} → {(1 - ref) * 525600:.0f} min/an")
for nom, valeur in (("3e serveur d'application", systeme(n_app=3)), ("auth doublée
    (99,9 % → 99,9999 %)", systeme(auth=1 - 0.001 ** 2)), ("basculement à 99,9 %",
    systeme(bascule=0.999))):
    print(f"{nom:36s} → {valeur:.5%} (+{(valeur - ref) * 100:.4f} points)")
assert abs(ref - 0.99890) < 0.0001
assert systeme(auth=1 - 0.001 ** 2) - ref > systeme(n_app=3) - ref > systeme(bascu
   le=0.999) - ref
```

Réponse (suite)

La disponibilité globale est d'environ **99,89 %** (≈ **9 heures** d'indisponibilité par an). Le **service d'authentification externe**, seul en série et à 99,9 %, est le maillon qui limite : doubler ce service apporte le plus. Le 3e serveur d'application apporte peu (la couche applicative est déjà à 99,9975 %), et fiabiliser le basculement presque rien (il ne sert que lors d'une panne du leader, événement rare). Raisonner par **maillon limitant** évite d'investir où le gain est faible.

À toi, en solo

Verrou distribué sûr. Implémente un service de verrous à baux avec jetons de barrière et un simulateur d'exécution avec pauses :

1. `acquerir(client, t)` renvoie un jeton croissant ; `renouveler(client, jeton, t)` prolonge le bail seulement si le jeton est encore valable ;
2. une ressource protégée qui applique la barrière ;
3. 3 clients qui, chacun, répètent : acquérir, lire la ressource (en présentant leur jeton), **pause aléatoire** (parfois plus longue que le bail), écrire `valeur + 1` avec leur jeton ;
4. avec un ordonnanceur à graine fixée, sur 300 exécutions : **sans** barrière, compte les mises à jour perdues (valeur finale < nombre d'écritures acceptées) ; **avec** barrière, vérifie que chaque écriture acceptée est basée sur la valeur la plus récente (aucune mise à jour perdue) ;
5. explique ce que la barrière ne garantit pas (une écriture refusée doit être réessayée par le client, qui peut ne jamais réussir si les pauses sont trop longues : vivacité).

À toi, en solo · réponse

Pistes de correction

```
import random

class Bail:
    def __init__(self): self.jeton, self.detenteur, self.expire = 0, None, -1
    def acquerir(self, client, t, duree=10):
        if self.detenteur is None or t >= self.expire:
            self.jeton += 1; self.detenteur, self.expire = client, t + duree; retu
   rn self.jeton
        return None

class Ressource:
    def __init__(self, barriere): self.barriere, self.max_jeton, self.valeur, self
   .acceptees, self.refusees = barriere, 0, 0, 0, 0
    def lire(self, jeton):                                    # la barrière
   couvre AUSSI la lecture qui précède l'écriture
        if self.barriere:
            if jeton < self.max_jeton: return None
            self.max_jeton = jeton
        return self.valeur
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def ecrire(self, jeton, nouvelle):
        if self.barriere and jeton < self.max_jeton: self.refusees += 1; return Fa
   lse
        self.max_jeton = max(self.max_jeton, jeton); self.valeur = nouvelle; self.
   acceptees += 1; return True

def simuler(barriere, graine, clients=3, tours=12):
    rng = random.Random(graine); bail, res = Bail(), Ressource(barriere)
    etat = {c: {"phase": "acquerir", "jeton": None, "lu": None, "reveil": 0} for c
    in range(clients)}
    t = 0
    while t < 400 and sum(1 for c in etat if etat[c]["phase"] != "fini") > 0:
        t += 1
        for c in rng.sample(list(etat), len(etat)):
            e = etat[c]
            if e["phase"] == "fini" or e["reveil"] > t: continue
            if e["phase"] == "acquerir":
                j = bail.acquerir(c, t)
                if j:
```

À toi, en solo · réponse

Pistes de correction (suite)

```
                    lu = res.lire(j)
                    if lu is not None: e.update(jeton=j, lu=lu, phase="ecrire", re
   veil=t + rng.choice([1, 2, 3, 25]))     # parfois une longue pause (25 > bail)
            elif e["phase"] == "ecrire":
                res.ecrire(e["jeton"], e["lu"] + 1)                    # écrit une
    valeur calculée à partir d'une lecture éventuellement PÉRIMÉE
                e["phase"] = "acquerir" if rng.random() < 0.6 else "fini"
    return res

perdues_sans = perdues_avec = 0
for g in range(300):
    for avec in (False, True):
        r = simuler(avec, g)
        perdu = r.acceptees - r.valeur                                  # écriture
   s acceptées dont l'effet a été écrasé
        if avec: perdues_avec += max(perdu, 0)
        else: perdues_sans += max(perdu, 0)
print(f"mises à jour perdues sur 300 exécutions : sans barrière {perdues_sans}, av
   ec barrière {perdues_avec}")
```

À toi, en solo · réponse

Pistes de correction (suite)

```
assert perdues_sans > 0 and perdues_avec == 0
```

Remarque importante : la barrière doit couvrir **aussi la lecture** qui précède l'écriture. Sinon un ancien détenteur (jeton 2) peut écrire entre la lecture du nouveau détenteur (jeton 3) et son écriture : la ressource n'a pas encore vu le jeton 3, accepte l'écriture périmée, et la mise à jour du nouveau détenteur repose sur une lecture devenue fausse. La barrière garantit la **sûreté** (aucune mise à jour perdue par un détenteur périmé) mais pas la **vivacité** : un client très lent peut voir toutes ses écritures refusées. Il doit alors **relire et réessayer** avec un nouveau bail. La sûreté vient de la ressource, la vivacité de la bonne volonté du réseau.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Croire qu'un bail non expiré à l'instant de la lecture est encore valide à l'écriture	Écritures d'un ancien détenteur	Jeton de barrière vérifié par la ressource
Détecteur de pannes à délai court sans confirmation	Basculements inutiles, deux leaders	Délai adapté, plusieurs observateurs, adaptatif
2PC entre services indépendants	Blocage des verrous si le coordinateur tombe	Sagas avec compensations idempotentes
Compensation non idempotente ou jamais réessayée	Remboursements en double ou oubliés	Idempotence + réessai jusqu'au succès

Mini-boss

Mini-boss

Matrice de pannes de HashBox. Prends l'architecture du service d'inscription (équilibreur, API ×3, paiement externe, base leader+réplica, file de messages, cache) et :

1. calcule sa disponibilité globale ; classe les composants par contribution à l'indisponibilité ;
2. écris la **matrice des pannes** : pour chaque composant, mode de panne, détection (délai, faux positifs), impact utilisateur, mécanisme de repli, procédure de reprise ;
3. implémente la saga d'inscription (place → paiement → notification → statistiques) avec orchestrateur durable (reprise après plantage), compensations idempotentes, étape pivot ;
4. injecte exhaustivement une panne à chaque point (avant/après chaque étape et chaque compensation) et vérifie l'invariant : à la fin, soit tout est fait une fois, soit rien n'est fait ;
5. implémente un verrou à bail avec jeton de barrière pour la modification concurrente d'un cours, et un test où un client est « mis en pause » ;
6. rédige un plan d'exercice de panne (« jour de jeu ») : 5 scénarios, hypothèse, mesure, critère de réussite.

Mini-boss · réponse

Pistes de correction

Un bon travail : calcul de disponibilité avec groupes parallèles et séries, tableau classant le paiement externe en tête; saga avec journal durable (étape courante + états de compensation), test exhaustif d'injection (n étapes × 3 points); verrou avec jetons croissants vérifiés par la ressource; plan d'exercice précis (par exemple « tuer le leader de la base pendant 1 000 inscriptions/min : hypothèse : < 5 erreurs, rétablissement < 30 s »).

Quiz éclair

Quiz éclair

1. Un jeton de barrière (fencing token) empêche :

A. la perte de paquets
B. l'écriture d'un ancien détenteur d'un bail expiré
C. les doublons de messages

2. Dans la validation en deux phases, si le coordinateur tombe après la préparation :

A. tout est automatiquement annulé
B. tout est automatiquement validé
C. les participants prêts restent bloqués, verrous tenus

3. Dans une saga, les compensations s'exécutent :

A. dans l'ordre inverse des étapes déjà faites, et doivent être idempotentes

Quiz éclair

Quiz éclair (suite)

B. dans le même ordre, une seule fois

C. sans ordre

4. 🔁 *Révision mission 08* : Valider l'offset après le traitement d'un message donne la sémantique :

A. au plus une fois

B. au moins une fois

C. exactement une fois garantie par le réseau

5. 🔁 *Révision mission 08* : Le patron outbox sert à :

A. chiffrer les messages

B. accélérer le courtier

C. publier un événement de façon atomique avec la modification en base

Quiz éclair · réponse

Réponses

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

Je réfléchis

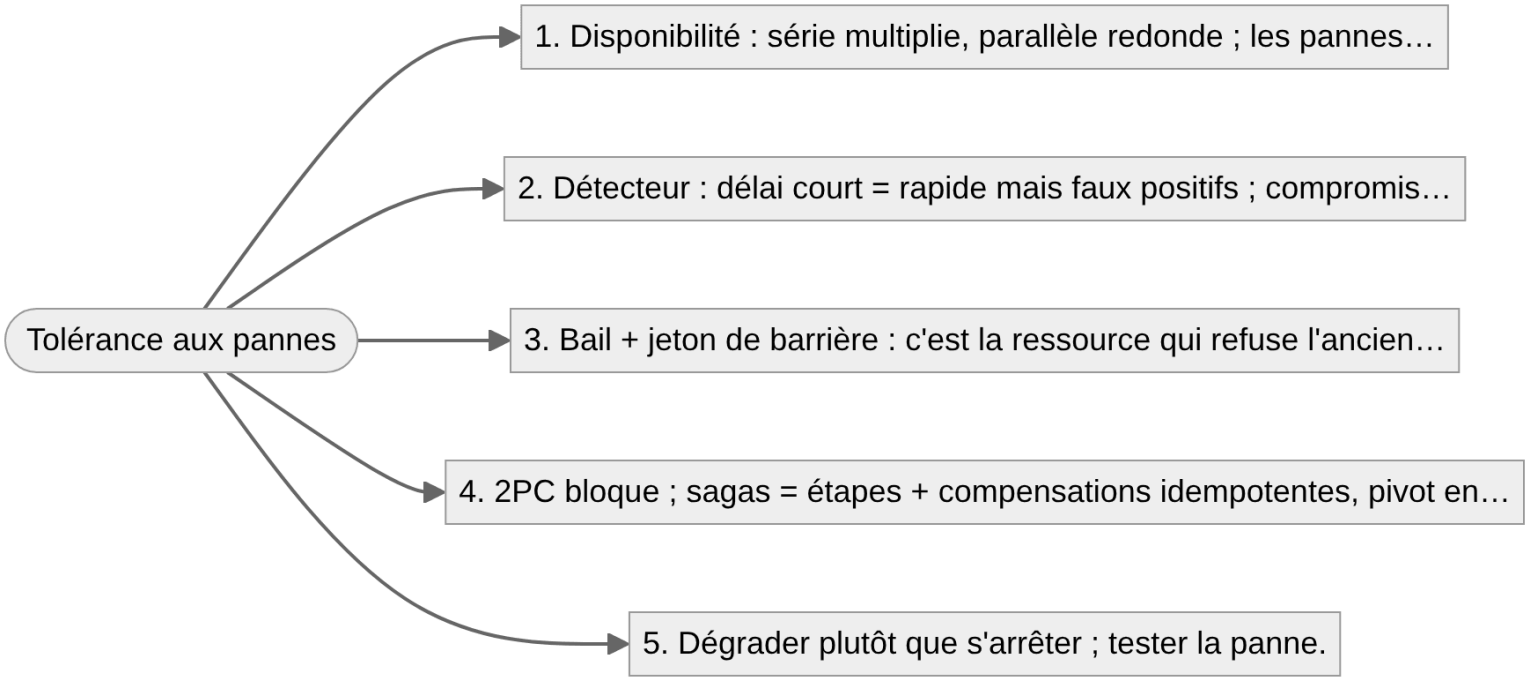
1. Quel composant de ton système, s'il tombait, ne serait remarqué par personne avant les utilisateurs ?
2. Quelle opération multi-services de ton système est « tout ou rien » sans que personne n'ait écrit comment on annule ?

À retenir

1. Disponibilité : série multiplie, parallèle redonde ; les pannes réelles ne sont pas indépendantes.
2. Détecteur : délai court = rapide mais faux positifs ; compromis explicite.
3. Bail + jeton de barrière : c'est la ressource qui refuse l'ancien détenteur.
4. 2PC bloque ; sagas = étapes + compensations idempotentes, pivot en dernier.
5. Dégrader plutôt que s'arrêter ; tester la panne.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Bâtisseur de résilience

Prochaine mission : Conception d'un service distribué

