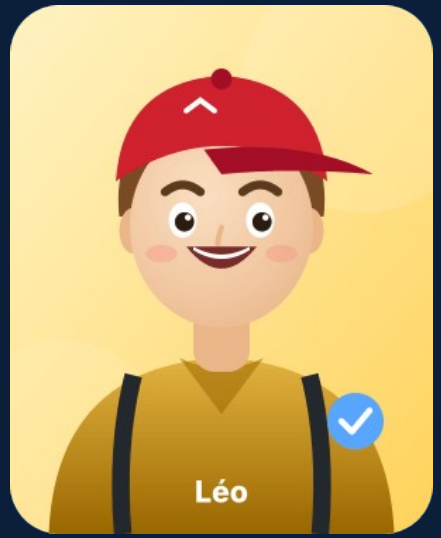
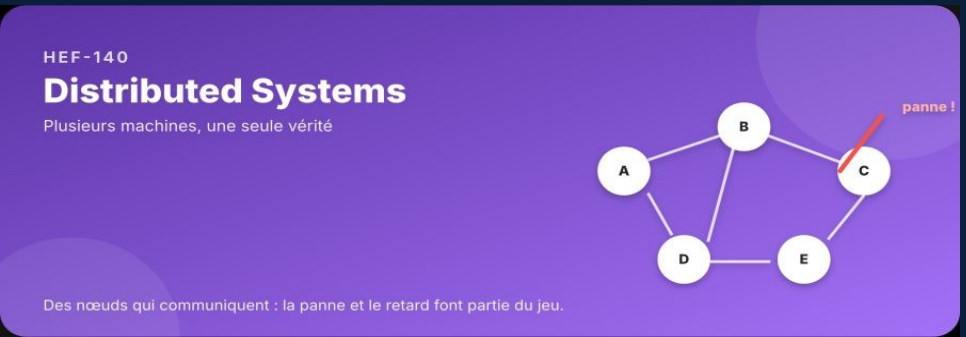
HEF-140 · MISSION 08 / 10

Files et événements



Remplacer les appels directs par des messages découple les services dans le temps — au prix de nouvelles règles : doublons, ordre, reprise, et un état qu'on reconstruit à partir de l'histoire.

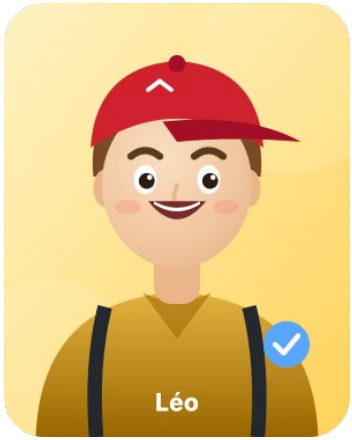


Badge : Facteur du système

110 min · avancé

L'histoire

Quand un formateur de HashBox publie un cours, six choses doivent arriver : indexer la recherche, prévenir les abonnés, mettre à jour les statistiques, générer le PDF, alerter la modération, mettre à jour le cache. L'API les appelle l'une après l'autre. Quand le service des PDF ralentit, la publication entière prend 12 secondes. Quand il tombe, la publication échoue, alors que le cours *est* enregistré en base. Et si l'API plante entre deux appels, certains services ne sont jamais prévenus.



« On peut inverser la logique, propose Maya : l'API enregistre le cours et publie un événement "CoursPublié". Chaque service s'abonne et fait son travail à son rythme, y compris après une panne. Mais ce découplage ouvre trois questions : que se passe-t-il si le message est livré deux fois ? Si l'API plante entre l'enregistrement et la publication ? Et dans quel ordre les messages sont-ils traités ? »

L'histoire (suite)

Tu apprends à construire un journal d'événements partitionné avec groupes de consommateurs et offsets, à garantir la livraison *au moins une fois*, à utiliser le patron « boîte d'envoi » (outbox), à traiter les messages en échec (file morte), et à reconstruire l'état par rejeu d'événements.

Ta mission

À la fin, je serai capable de…

1. distinguer file de messages, publication/abonnement et journal d'événements
2. implémenter partitions, offsets et groupes de consommateurs avec livraison au moins une fois
3. utiliser une file morte (dead-letter) pour les messages en échec répété
4. appliquer le patron outbox pour publier un événement de façon atomique avec l'écriture en base
5. reconstruire un état par rejeu d'événements (event sourcing) et projeter des vues de lecture

Défi d'échauffement

Défi d'échauffement

Dans un restaurant, les serveurs écrivent des bons sur un passe-plat et la cuisine les prend dans l'ordre. Si la cuisine est débordée, les bons s'accumulent sans bloquer les serveurs. Si un cuisinier tombe malade, un autre reprend la pile. Si un bon est collé deux fois, le plat est préparé deux fois. Si un serveur écrit le bon… puis oublie de le mettre sur le passe-plat ? Liste les avantages et les risques de ce fonctionnement.

Le concept

Trois outils de messagerie

Outil	Principe	Usage
File de messages (*queue*)	chaque message est traité par **un seul** consommateur parmi plusieurs ; il disparaît une fois traité	répartir du travail
Publication/abonnement (*pub/sub*)	chaque abonné reçoit **sa copie** de chaque message	notifier plusieurs services
Journal d'événements (*log*, Kafka)	suite **ordonnée et persistante** de messages ; chaque consommateur lit à son rythme avec un **offset** ; rejouable	flux d'événements, reconstruction d'état

Le concept

Le journal partitionné

Un **sujet** (*topic*) est divisé en **partitions** (mission 6). Dans une partition :

- les messages sont **ordonnés** (offsets 0, 1, 2…) ;
- l'ordre n'est garanti **qu'au sein d'une partition**, pas entre partitions.

La **clé** du message (par exemple l'identifiant du cours) détermine la partition : tous les événements d'un même cours vont dans la même partition, donc **dans l'ordre**.

Groupes de consommateurs

Le concept

Groupes de consommateurs (suite)

Un **groupe** de consommateurs se partage les partitions : chaque partition est lue par **un seul** consommateur du groupe (parallélisme, ordre préservé). Plusieurs groupes distincts lisent indépendamment (chacun reçoit tous les messages). Quand un consommateur tombe ou arrive, les partitions sont **réaffectées** (*rebalance*).

Offsets et livraison

Le consommateur enregistre (**valide**, *commit*) son **offset** : « j'ai traité jusqu'ici ». Selon le moment de la validation :

Choix	Comportement en cas de panne	Sémantique
Valider **avant** de traiter	message perdu si panne pendant le traitement	au plus une fois
Valider **après** traitement	message **rejoué** si panne avant validation	**au moins une fois** (le standard)

Le concept

Offsets et livraison (suite)

D'où : consommateurs **idempotents** (mission 4).

Messages en échec : file morte

Un message qui échoue toujours (« message empoisonné ») bloquerait la partition. Après `N` tentatives (avec recul), on le déplace dans une **file morte** (*dead-letter queue*) pour examen humain, et on continue.

Le problème de l'écriture double

```
1. enregistrer le cours en base       ← réussi
2. publier l'événement CoursPublié    ← l'API plante ici : l'événement n'est jamai
   s publié !
```

Deux systèmes (base + courtier) ne se mettent pas à jour de façon atomique. **Patron « boîte d'envoi » (outbox)** :

Le concept

Le problème de l'écriture double (suite)

1. dans **la même transaction** que la modification, insérer l'événement dans une table `outbox` ;
2. un **relais** lit la table `outbox`, publie dans le courtier, puis marque l'événement comme publié ;
3. si le relais plante entre la publication et le marquage, l'événement est republié → **doublon** → consommateurs idempotents.

Résultat : *au moins une fois*, sans événement perdu.

Événements et ordre

- **Un événement décrit un fait passé** (« CoursPublié », « InscriptionAnnulée »), immuable, avec identifiant, horodatage, version.
- Ne pas supposer l'ordre **entre** partitions ; utiliser une **version** par entité pour ignorer un événement plus ancien que l'état.
- **Évolution** des schémas d'événements : compatibilité ascendante (mission 2).

Le concept

Sourçage d'événements (*event sourcing*)

Au lieu de stocker l'**état courant**, on stocke la **suite des événements** qui y ont mené. L'état est **déduit par rejeu** (`plier` les événements).

Avantage	Coût
historique complet, audit	complexité
« voyager dans le temps » : état à n'importe quelle date	rejeu long : on prend des **instantanés** (*snapshots*)
nouvelles vues sans migration : on rejoue	évolution des événements

CQRS et projections

Le concept

CQRS et projections (suite)

CQRS sépare le modèle d'**écriture** (commandes, événements) des modèles de **lecture** (vues optimisées pour les requêtes), appelés **projections**, alimentés par les événements. Les vues sont **finalement cohérentes** : un léger retard est normal.

Contre-pression et dimensionnement

Une file qui grandit signale que les consommateurs sont trop lents : surveiller le **retard** (*lag* = offset du dernier message − offset validé). Remèdes : plus de consommateurs (jusqu'au nombre de partitions), traitement par lots, optimisation.

🧠 ***Astuce** : décris chaque flux par trois phrases : qui produit, qui consomme, que se passe-t-il si le message arrive deux fois ou dans le désordre ?*

Exemple pas à pas

Problème : construire un courtier en mémoire avec partitions, offsets et groupes de consommateurs, montrer la livraison au moins une fois après panne, la file morte, le patron outbox avec un relais qui plante, puis reconstruire un état par rejeu d'événements.

```
import hashlib, json, sqlite3
from collections import defaultdict

# ---------------- 1. un courtier : journal partitionné, offsets par groupe ------
   ----------
class Courtier:
    def __init__(self, partitions=3):
        self.n = partitions
        self.journal = [[] for _ in range(partitions)]                          # cha
   que partition : liste ordonnée de messages
        self.offsets = defaultdict(lambda: [0] * partitions)                    # off
   set VALIDÉ par groupe et par partition
    def publier(self, cle, valeur):
        p = int(hashlib.md5(str(cle).encode()).hexdigest(), 16) % self.n      # mê
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    me clé → même partition → ordre préservé
         self.journal[p].append({"cle": cle, "valeur": valeur}); return p, len(self
    .journal[p]) - 1
     def lire(self, groupe, partition, max_messages=100):
         debut = self.offsets[groupe][partition]
         return [(debut + i, m) for i, m in enumerate(self.journal[partition][debut
    :debut + max_messages])]
     def valider(self, groupe, partition, offset):                              # « j
    'ai traité jusqu'à `offset` inclus »
         self.offsets[groupe][partition] = max(self.offsets[groupe][partition], off
    set + 1)
     def retard(self, groupe): return sum(len(self.journal[p]) - self.offsets[group
    e][p] for p in range(self.n))

courtier = Courtier(3)
for i in range(30):
    courtier.publier(f"cours:{i % 5}", {"evenement": "inscription", "n": i, "cours
    ": f"cours:{i % 5}"})
# l'ordre est préservé PAR CLÉ
```

Exemple pas à pas (suite)

```
for cle in ("cours:0", "cours:3"):
    p = int(hashlib.md5(cle.encode()).hexdigest(), 16) % 3
    ns = [m["valeur"]["n"] for m in courtier.journal[p] if m["cle"] == cle]
    assert ns == sorted(ns)
print("retard du groupe « stats » avant lecture :", courtier.retard("stats"))
assert courtier.retard("stats") == 30

# ---------------- 2. livraison AU MOINS UNE FOIS : panne avant validation → rejeu
    ----------------
def consommer(groupe, courtier, traiter, planter_a=None, lot=5):
    traites = 0
    for p in range(courtier.n):
        for offset, m in courtier.lire(groupe, p, lot):
            traiter(m["valeur"]); traites += 1
            if planter_a is not None and traites == planter_a: raise RuntimeError(
   "panne du consommateur AVANT la validation du lot")
        if courtier.journal[p]: courtier.valider(groupe, p, courtier.offsets[group
   e][p] + min(lot, len(courtier.journal[p]) - courtier.offsets[groupe][p]) - 1)
    return traites
```

Exemple pas à pas (suite)

```
vus = []
try: consommer("stats", courtier, vus.append, planter_a=7)
except RuntimeError as e: print("panne :", e)
avant = len(vus)
while courtier.retard("stats") > 0: consommer("stats", courtier, vus.append)
doublons = len(vus) - len({m["n"] for m in vus})
print(f"messages traités {len(vus)} pour 30 publiés → {doublons} traitements en do
   uble (livraison au moins une fois)")
assert {m["n"] for m in vus} == set(range(30)) and doublons > 0 and courtier.retar
   d("stats") == 0

# ---------------- 3. consommateur idempotent : plus aucun effet en double -------
   ---------
effets, deja = defaultdict(int), set()
def traiter_idempotent(valeur):
    if valeur["n"] in deja: return
    deja.add(valeur["n"]); effets[valeur["cours"]] += 1
c2 = Courtier(3)
```

Exemple pas à pas (suite)

```
for i in range(30): c2.publier(f"cours:{i % 5}", {"n": i, "cours": f"cours:{i % 5}
   "})
try: consommer("g", c2, traiter_idempotent, planter_a=7)
except RuntimeError: pass
while c2.retard("g") > 0: consommer("g", c2, traiter_idempotent)
assert sum(effets.values()) == 30 and all(v == 6 for v in effets.values())
print("avec idempotence : 30 messages, 30 effets, malgré la panne")

# ---------------- 4. file morte : un message empoisonné ne bloque pas la partitio
   n ----------------
def traiter_avec_morte(courtier, groupe, partition, traiter, max_essais=3):
    morte, essais = [], defaultdict(int)
    while courtier.retard(groupe) > 0:
        for offset, m in courtier.lire(groupe, partition, 10):
            while True:
                try: traiter(m["valeur"]); break
                except ValueError:
                    essais[offset] += 1
                    if essais[offset] >= max_essais: morte.append(m); break
```

Exemple pas à pas

Exemple pas à pas (suite)

```
      # trop d'échecs : file morte
            courtier.valider(groupe, partition, offset)
        if all(courtier.offsets[groupe][p] >= len(courtier.journal[p]) for p in [p
   artition]): break
    return morte
c3 = Courtier(1)
for i in range(6): c3.publier("k", {"n": i, "poison": i == 2})
ok = []
def fragile(v):
    if v["poison"]: raise ValueError("message mal formé")
    ok.append(v["n"])
morte = traiter_avec_morte(c3, "g", 0, fragile)
print("traités :", ok, "| file morte :", [m["valeur"]["n"] for m in morte])
assert ok == [0, 1, 3, 4, 5] and [m["valeur"]["n"] for m in morte] == [2]

# ---------------- 5. patron « outbox » : l'événement est écrit avec la donnée, da
   ns la MÊME transaction ----------------
class Service:
    def __init__(self):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        self.bd = sqlite3.connect(":memory:", isolation_level=None)
        self.bd.executescript("CREATE TABLE cours (id INTEGER PRIMARY KEY, titre T
    EXT); CREATE TABLE outbox (id INTEGER PRIMARY KEY AUTOINCREMENT, evenement TEXT
    , publie INTEGER DEFAULT 0);")
     def publier_cours(self, titre, planter=False):
        self.bd.execute("BEGIN")
        try:
            cur = self.bd.execute("INSERT INTO cours (titre) VALUES (?)", (titre,)
    )
            self.bd.execute("INSERT INTO outbox (evenement) VALUES (?)", (json.dum
    ps({"type": "CoursPublié", "id": cur.lastrowid, "titre": titre}),))
            if planter: raise RuntimeError("panne avant COMMIT")
            self.bd.execute("COMMIT")
        except RuntimeError:
            self.bd.execute("ROLLBACK"); raise
     def relais(self, courtier, planter_apres_publication=False):
        for id_, ev in self.bd.execute("SELECT id, evenement FROM outbox WHERE pub
    lie = 0 ORDER BY id").fetchall():
            courtier.publier("cours", json.loads(ev))
```

Exemple pas à pas (suite)

```
            if planter_apres_publication: raise RuntimeError("le relais plante APR
   ÈS la publication, AVANT le marquage")
            self.bd.execute("UPDATE outbox SET publie = 1 WHERE id = ?", (id_,))

s, cb = Service(), Courtier(1)
try: s.publier_cours("Algorithmes", planter=True)
except RuntimeError: pass
assert s.bd.execute("SELECT COUNT(*) FROM cours").fetchone()[0] == 0 and s.bd.exec
   ute("SELECT COUNT(*) FROM outbox").fetchone()[0] == 0      # ni donnée, ni évén
   ement : atomique
s.publier_cours("Algorithmes")
try: s.relais(cb, planter_apres_publication=True)
except RuntimeError as e: print("relais :", e)
s.relais(cb)
                # le relais redémarre : republie
evts = cb.journal[0]
print("événements dans le courtier :", len(evts), "(un doublon : le consommateur d
   oit être idempotent)")
assert len(evts) == 2 and evts[0] == evts[1]
```

Exemple pas à pas (suite)

```
# ---------------- 6. sourçage d'événements : l'état est un REJEU ----------------
EVENEMENTS = [("CoursCree", {"cours": "HEF-010", "places": 3}), ("Inscrit", {"cour
   s": "HEF-010", "membre": "ada"}), ("Inscrit", {"cours": "HEF-010", "membre": "b
   ob"}),
              ("Desinscrit", {"cours": "HEF-010", "membre": "ada"}), ("PlacesAugme
   ntees", {"cours": "HEF-010", "places": 5}), ("Inscrit", {"cours": "HEF-010", "m
   embre": "cleo"})]
def appliquer(etat, evenement):
    type_, d = evenement
    etat = {k: (set(v) if isinstance(v, set) else v) for k, v in etat.items()}
    if type_ == "CoursCree": etat.update(places=d["places"], inscrits=set())
    elif type_ == "Inscrit": etat["inscrits"].add(d["membre"])
    elif type_ == "Desinscrit": etat["inscrits"].discard(d["membre"])
    elif type_ == "PlacesAugmentees": etat["places"] = d["places"]
    return etat
def rejouer(evenements, jusqua=None):
    etat = {}
    for e in evenements[:jusqua]: etat = appliquer(etat, e)
```

Exemple pas à pas (suite)

```
    return etat
final = rejouer(EVENEMENTS)
print("état final :", final, "| état après 3 événements :", rejouer(EVENEMENTS, 3)
   )
assert final == {"places": 5, "inscrits": {"bob", "cleo"}} and rejouer(EVENEMENTS,
    3)["inscrits"] == {"ada", "bob"}
```

Lecture :

1. **L'ordre est préservé par clé** (dans une partition) ; entre partitions, aucun ordre n'est garanti.
2. En validant l'offset **après** traitement, une panne provoque un **rejeu** : tous les messages sont traités, **certains deux fois** (*au moins une fois*).
3. Avec un **consommateur idempotent**, les doublons restent sans effet : 30 messages, 30 effets.
4. La **file morte** isole un message empoisonné après 3 essais ; la partition continue.

Exemple pas à pas (suite)

5. Le patron **outbox** rend l'enregistrement et la publication de l'événement atomiques ; le relais peut republier après une panne (doublon, absorbé par l'idempotence).
6. L'état d'un cours se **reconstruit par rejeu** des événements ; on peut aussi « voyager dans le temps » (état après 3 événements).

À toi, avec aide

Écris `affecter(partitions, consommateurs)` qui répartit `P` partitions entre `C` consommateurs d'un même groupe (affectation par plages contiguës, aussi équilibrée que possible), puis simule le **rééquilibrage** quand un consommateur disparaît. Vérifie que :

1. chaque partition est affectée à **exactement un** consommateur ;
2. l'écart de charge entre consommateurs est au plus 1 ;
3. avec 6 partitions et 3 consommateurs → 2 chacun ; après la perte d'un consommateur → 3 et 3 ;
4. avec plus de consommateurs que de partitions, certains restent inactifs.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Chaque consommateur reçoit `P // C` partitions, les `P % C` premiers en reçoivent une de plus.

Indice 2

Retire le consommateur et recalcule.

Indice 3

Compte les partitions par consommateur avec un `Counter`.

Réponse

```
from collections import Counter

def affecter(nb_partitions, consommateurs):
    consommateurs = sorted(consommateurs)
    base, reste = divmod(nb_partitions, len(consommateurs))
    attribution, debut = {}, 0
    for i, c in enumerate(consommateurs):
        taille = base + (1 if i < reste else 0)
        attribution[c] = list(range(debut, debut + taille)); debut += taille
    return attribution

a = affecter(6, ["c1", "c2", "c3"])
print(a)
assert all(len(v) == 2 for v in a.values())
b = affecter(6, ["c1", "c3"])                                  # c2 a disparu
print("après la perte de c2 :", b)
assert all(len(v) == 3 for v in b.values())
for nb, cs in ((7, ["a", "b", "c"]), (10, list("abcd")), (3, list("abcde"))):
    r = affecter(nb, cs)
```

À toi, avec aide · réponse

Réponse (suite)

```
    toutes = sorted(p for v in r.values() for p in v)
    assert toutes == list(range(nb))                          # chaque partition af
   fectée exactement une fois
    tailles = [len(v) for v in r.values()]; assert max(tailles) - min(tailles) <=
   1
assert sorted(len(v) for v in affecter(3, list("abcde")).values()) == [0, 0, 1, 1,
    1]       # 2 consommateurs inactifs
```

Le **parallélisme maximal** d'un groupe est le nombre de partitions : au-delà, les consommateurs en plus restent inactifs. C'est pourquoi on crée plus de partitions que de consommateurs prévus.

À toi, en solo

Sourçage d'événements avec instantanés et projection. Pour un agrégat « cours » (créer, inscrire, désinscrire, augmenter les places, fermer) :

1. un magasin d'événements `append(agregat, evenement, version_attendue)` qui refuse l'écriture si la version attendue ne correspond pas (**concurrence optimiste**) ;
2. `etat(agregat)` rejoue les événements, avec un **instantané** tous les 5 événements (rejoue à partir du dernier instantané) ;
3. une **commande** `inscrire(cours, membre)` qui valide la règle métier (places disponibles, cours ouvert, pas de doublon) en rejouant l'état, et refuse sinon ;
4. une **projection** « liste des cours complets » alimentée par les événements (mise à jour incrémentale) ;
5. teste : (a) l'état par rejeu complet égale l'état avec instantanés pour 200 événements aléatoires ; (b) deux commandes concurrentes sur la même version : l'une est refusée ; (c) la projection égale un calcul complet par rejeu.

À toi, en solo · réponse

Pistes de correction

```
import random, copy

class ConflitVersion(Exception): pass

def appliquer(etat, evt):
    etat = copy.deepcopy(etat); t, d = evt
    if t == "Cree": etat = {"places": d["places"], "inscrits": [], "ferme": False}
    elif t == "Inscrit": etat["inscrits"].append(d["membre"])
    elif t == "Desinscrit": etat["inscrits"].remove(d["membre"])
    elif t == "Places": etat["places"] = d["places"]
    elif t == "Ferme": etat["ferme"] = True
    return etat

class Magasin:
    def __init__(self, intervalle=5):
        self.evenements, self.instantanes, self.intervalle = {}, {}, intervalle
        self.abonnes = []
    def version(self, agregat): return len(self.evenements.get(agregat, []))
    def append(self, agregat, evt, version_attendue):
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        if self.version(agregat) != version_attendue: raise ConflitVersion(f"versi
    on {self.version(agregat)} au lieu de {version_attendue}")
        self.evenements.setdefault(agregat, []).append(evt)
        n = self.version(agregat)
        if n % self.intervalle == 0: self.instantanes[agregat] = (n, self.etat(agr
    egat, utiliser_instantane=False))
        for f in self.abonnes: f(agregat, evt)
     def etat(self, agregat, utiliser_instantane=True):
        debut, etat = 0, {}
        if utiliser_instantane and agregat in self.instantanes: debut, etat = self
    .instantanes[agregat]
        for evt in self.evenements.get(agregat, [])[debut:]: etat = appliquer(etat
    , evt)
        return etat

def inscrire(m, cours, membre):
    v, e = m.version(cours), m.etat(cours)
    if not e or e["ferme"]: raise ValueError("cours inexistant ou fermé")
    if membre in e["inscrits"]: raise ValueError("déjà inscrit")
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    if len(e["inscrits"]) >= e["places"]: raise ValueError("complet")
    m.append(cours, ("Inscrit", {"membre": membre}), v)

# projection : liste des cours complets, mise à jour incrémentale
complets = set()
def projeter(m):
    def f(agregat, evt):
        e = m.etat(agregat)
        (complets.add if e and len(e["inscrits"]) >= e["places"] else complets.dis
   card)(agregat)
    return f

m = Magasin(); m.abonnes.append(projeter(m))
rng = random.Random(1)
for c in ("c1", "c2", "c3"): m.append(c, ("Cree", {"places": 3}), 0)
for _ in range(200):
    c, membre = rng.choice(["c1", "c2", "c3"]), f"m{rng.randrange(8)}"
    action = rng.random()
    try:
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        if action < 0.6: inscrire(m, c, membre)
        elif action < 0.8 and membre in m.etat(c).get("inscrits", []): m.append(c,
    ("Desinscrit", {"membre": membre}), m.version(c))
        elif action < 0.9: m.append(c, ("Places", {"places": rng.randint(2, 5)}),
   m.version(c))
    except ValueError: pass
for c in ("c1", "c2", "c3"):
    assert m.etat(c) == m.etat(c, utiliser_instantane=False)          # (a) insta
   ntané ≡ rejeu complet
    assert (c in complets) == (len(m.etat(c)["inscrits"]) >= m.etat(c)["places"])
         # (c) projection ≡ calcul complet
# (b) concurrence optimiste
v = m.version("c1")
m.append("c1", ("Places", {"places": 10}), v)
try: m.append("c1", ("Places", {"places": 20}), v); raise AssertionError
except ConflitVersion as e: print("deuxième commande concurrente refusée :", e)
print("sourçage d'événements : instantanés et projection cohérents avec le rejeu c
   omplet")
```

À toi, en solo · réponse

Pistes de correction (suite)

L'**instantané** n'est qu'une optimisation : il doit toujours être **équivalent** au rejeu complet (c'est le test central). La **version attendue** empêche deux commandes concurrentes d'écraser leurs décisions respectives (concurrence optimiste).

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Écrire en base puis publier, sans outbox	Événement perdu si panne entre les deux	Outbox transactionnelle + relais
Valider l'offset avant de traiter	Messages perdus en cas de panne	Valider après traitement + consommateur idempotent
Supposer l'ordre global	Événements traités dans le désordre	Ordre par clé/partition, versions par entité
Message empoisonné qui bloque la partition	Retard infini	Nombre d'essais borné, file morte, alerte

Mini-boss

Mini-boss

Pipeline de commandes de HashBox. Concois et simule : un service `Inscriptions` (base SQLite + outbox), un courtier à 3 partitions, trois consommateurs (`Paiement`, `Notifications`, `Statistiques`) dans des groupes distincts.

1. outbox atomique, relais avec injection de panne à chaque point ;
2. consommateurs avec boîte de réception (messages traités) atomique avec leurs effets ;
3. file morte avec recul, rejouable après correction du bug ;
4. injection de pannes (relais, consommateur avant/après validation, perte de connexion), duplication et réordonnancement entre partitions ;
5. invariants vérifiés sur 200 graines : aucune inscription sans événement, aucun événement sans inscription, chaque événement a **un effet** dans chaque consommateur, jamais deux paiements pour la même inscription ;
6. mesure du retard (lag) et du temps de convergence après panne ;

Mini-boss

Mini-boss (suite)

7. documente le schéma d'événements et sa stratégie d'évolution.

Mini-boss · réponse

Pistes de correction

Un bon travail : tables `inscriptions`, `outbox`, `boite_reception(consommateur, evenement_id)` par consommateur; transactions locales atomiques; file morte avec compteur d'essais; test d'injection exhaustive des points de panne; invariants vérifiés après convergence; schéma d'événement `{id, type, version_schema, entite, version, donnees}` avec règles d'évolution (champs optionnels ajoutés, jamais retirés).

Quiz éclair

Quiz éclair

1. Valider l'offset après le traitement d'un message donne la sémantique :

A. au plus une fois
B. exactement une fois garantie par le réseau
C. au moins une fois

2. Le patron outbox sert à :

A. publier un événement de façon atomique avec la modification en base
B. chiffrer les messages
C. accélérer le courtier

3. Dans un journal partitionné, l'ordre est garanti :

A. entre toutes les partitions

Quiz éclair

Quiz éclair (suite)

B. au sein d'une partition

C. nulle part

4. 🔁 *Révision mission 07* : Pendant une partition réseau, un système CP :

A. répond partout avec des copies divergentes

B. ignore la partition

C. refuse de répondre côté minoritaire pour préserver la cohérence

5. 🔁 *Révision mission 07* : Dans Raft, une entrée est validée (committed) quand elle est stockée sur :

A. une majorité de nœuds

B. tous les nœuds

C. un seul nœud

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quel appel synchrone de ton système gagnerait à devenir un événement, et que perdrais-tu ?
2. Que se passe-t-il chez toi si un consommateur reçoit le même message deux fois ?

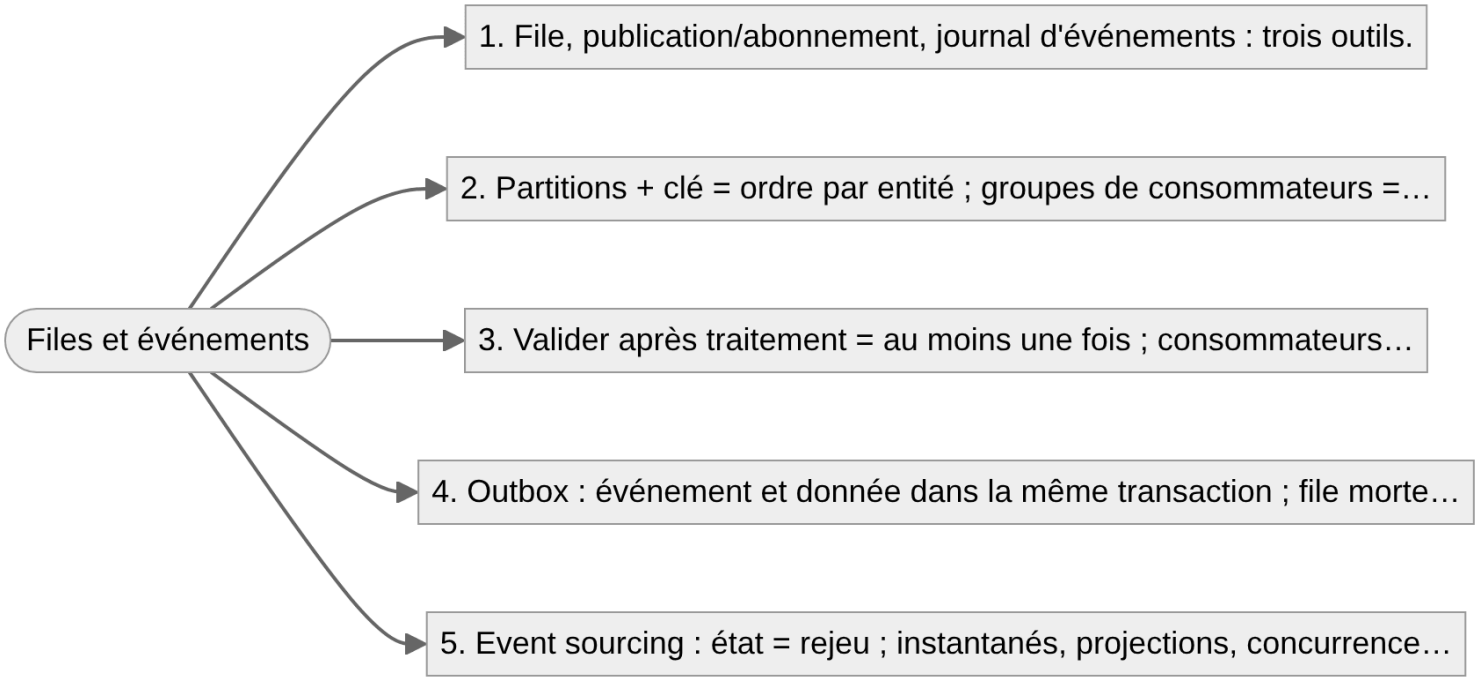
À retenir

À retenir

1. File, publication/abonnement, journal d'événements : trois outils.
2. Partitions + clé = ordre par entité ; groupes de consommateurs = parallélisme.
3. Valider après traitement = au moins une fois ; consommateurs idempotents.
4. Outbox : événement et donnée dans la même transaction ; file morte pour l'empoisonné.
5. Event sourcing : état = rejeu ; instantanés, projections, concurrence optimiste.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Facteur du système

Prochaine mission : Tolérance aux pannes

