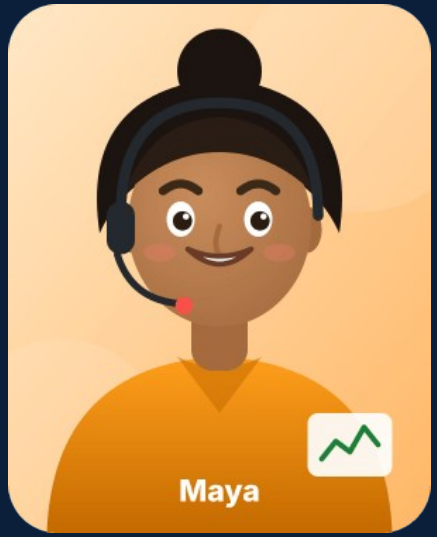
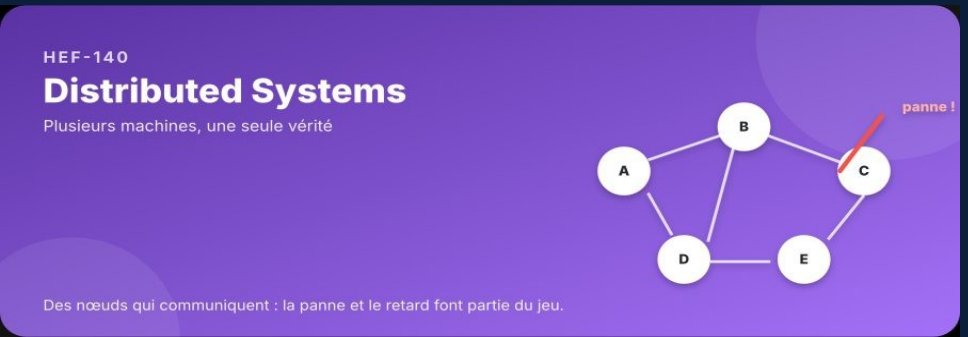
HEF-140 · MISSION 10 / 10

Conception d'un service distribué

Assembler partitionnement, réplication, idempotence, messages et reprise sur panne en un service dont on peut démontrer les garanties — par simulation, injection de pannes et invariants.



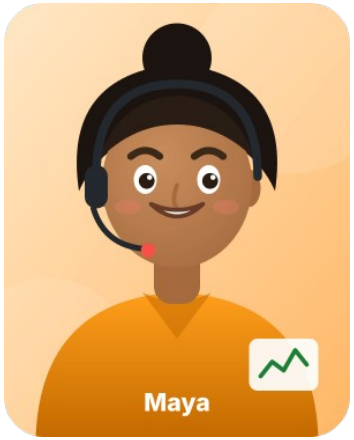


Badge : Architecte du distribué

125 min · avancé

L'histoire

HashBox lance les **inscriptions en direct** pour un événement de 5 000 personnes. Chaque cours a 40 places, des milliers de personnes cliquent en même temps, des serveurs tombent, des réseaux coupent, des utilisateurs cliquent deux fois. Maya pose les exigences : « Jamais plus de 40 inscrits. Jamais deux fois la même personne. Jamais une inscription confirmée qui disparaît. Et si possible, le service reste disponible. »



> *« Quatre invariants, dit Sam. On ne les garantira pas en espérant. On les garantit en choisissant des mécanismes précis — et on le vérifie par simulation, en injectant des pannes comme le ferait un adversaire. »*

Dans cette dernière mission, tu conçois un service distribué complet, tu écris ses invariants, et tu les mets à l'épreuve avec un banc d'essai de pannes aléatoires — où une conception naïve échoue et une conception réfléchie tient.

Ta mission

À la fin, je serai capable de…

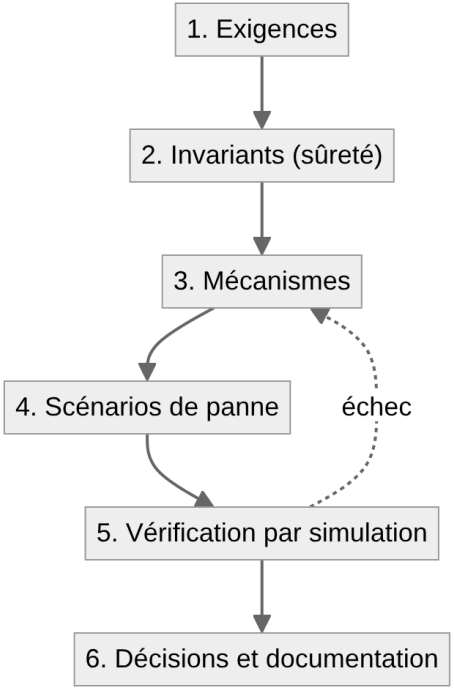
1. partir d'exigences pour fixer des invariants vérifiables (sûreté) et des objectifs de disponibilité (vivacité)
2. choisir des mécanismes (partitionnement, réplication, idempotence, échéances) justifiés par les invariants
3. simuler un service distribué avec réseau défaillant, doublons et réordonnancements
4. vérifier les invariants sur de nombreuses exécutions aléatoires (test à la Jepsen)
5. documenter l'architecture : diagrammes, contrats, scénarios de panne, décisions (ADR)

Défi d'échauffement

Défi d'échauffement

Tu organises une soirée avec 40 places et un système de réservation par SMS. Liste les façons dont cela peut mal tourner : deux personnes prennent la dernière place en même temps, quelqu'un envoie deux fois le même SMS parce que la réponse tarde, le téléphone du responsable tombe en panne. Pour chaque problème, quelle règle simple protégerait ?

Démarche de conception (suite)



Le concept

Démarche de conception (suite)

Description du schéma : six étapes : exigences, invariants, mécanismes, scénarios de panne, vérification par simulation, décisions et documentation ; un échec à la vérification renvoie à la conception des mécanismes.

1. Des exigences aux invariants

Distinguer :

Type	Définition	Exemple
Sûreté (*safety*)	« quelque chose de mauvais n'arrive **jamais** »	jamais plus de 40 inscrits ; jamais d'inscription confirmée perdue
Vivacité (*liveness*)	« quelque chose de bon arrive **finalement** »	toute demande reçoit finalement une réponse
Qualité de service	niveau mesuré	99,9 % des inscriptions en moins de 500 ms

Le concept

1. Des exigences aux invariants (suite)

La **sûreté** s'énonce comme un **invariant** vérifiable sur n'importe quelle exécution. En cas de conflit pendant une panne, on **sacrifie la vivacité avant la sûreté** pour les données critiques (mission 7).

2. Du mécanisme à la garantie

Invariant	Menace	Mécanisme
Pas de surbooking	deux inscriptions concurrentes qui lisent la même place libre	**un seul écrivain** par cours (partition par cours, leader), opération « vérifier puis agir » atomique
Pas de doublon d'intention	relance après réponse perdue, double-clic	**clé d'idempotence** + réponse mémorisée
Pas de perte d'une inscription confirmée	panne du leader	**réplication** à la majorité avant confirmation, élection à majorité

2. Du mécanisme à la garantie (suite)

Invariant	Menace	Mécanisme
Disponibilité	panne d'un nœud	réplicas + basculement ; clients avec **délai, relances bornées, échéance**

La conception est la **table qui relie chaque menace à un mécanisme**. Chaque ligne doit être testée.

3. Les scénarios de panne

Pour chaque composant et chaque message : **perdu, dupliqué, retardé, réordonné** ; nœud : **arrêté, redémarré, ralenti, isolé** ; client : **abandonne, relance, clique deux fois**.

4. Vérifier par simulation

Un **banc d'essai déterministe** :

4. Vérifier par simulation (suite)

1. modèle du système (processus, réseau avec fautes configurables) ;
2. **graine** pour la reproductibilité : un échec se rejoue à l'identique ;
3. génération de **charges et fautes aléatoires** ;
4. **invariants** vérifiés à la fin (et au fil de l'eau) ;
5. des centaines/milliers de graines ;
6. **minimisation** : réduire un scénario en échec au plus petit qui échoue encore.

C'est la méthode de la simulation déterministe (FoundationDB) et des tests Jepsen. Elle révèle des bugs de **concurrence et de panne** que les tests classiques ne voient presque jamais.

5. Documenter

Le concept

5. Documenter (suite)

Document	Contenu
Diagramme d'architecture	composants, flux, frontières de confiance
Contrats	API, événements, erreurs, idempotence
Invariants et mécanismes	la table menace → mécanisme → test
Scénarios de panne	hypothèses, comportement attendu, résultat des essais
ADR (registre des décisions)	contexte, options, décision, conséquences
Exploitation	indicateurs, alertes, procédures

🧠 ***Astuce*** *: une conception distribuée n'est pas « correcte » parce qu'elle est élégante : elle l'est parce que tu as essayé de la casser, avec méthode, et qu'elle a tenu.*

Exemple pas à pas

Problème : trois conceptions du service d'inscription en direct, comparées par un banc d'essai de pannes (réseau qui perd les requêtes et les réponses, duplique les messages, clients qui relancent, exécutions entrelacées). Invariants : ① jamais plus de 40 inscrits, ② jamais deux inscriptions pour une même personne, ③ toute inscription confirmée est présente.

```
import random

CAPACITE = 5                                                    # réduit à 5
    places pour que la ruée soit sévère

class Reseau:
    # Chaque appel peut : perdre la requête, dupliquer la requête, perdre la répon
   se.
    def __init__(self, rng, p_perte_req=0.15, p_dup=0.15, p_perte_rep=0.15):
        self.rng, self.pr, self.pd, self.pp = rng, p_perte_req, p_dup, p_perte_rep

# ---------------- le service : trois conceptions ----------------
```

Exemple pas à pas (suite)

```
class Service:
    def __init__(self, conception):
        self.c = conception
        self.inscrits = []                                          # liste de (
   membre, cle)
        self.reponses = {}                                          # clé d'idem
   potence → réponse mémorisée
        self.replique = []                                          # copie sur
   un second nœud (durabilité)
    # --- conception A : naïve (lire puis écrire, sans clé) : la lecture et l'écri
   ture sont séparées par un point d'entrelacement ---
    def inscrire_A(self, membre, cle):
        n = len(self.inscrits)
        yield                                                        # un autre
   client peut s'exécuter ici
        if n < CAPACITE:
            self.inscrits.append((membre, cle)); return "confirmé"
        return "complet"
    # --- conception B : A + clé d'idempotence (le doublon est ignoré), toujours n
```

Exemple pas à pas (suite)

```
on atomique ---
 def inscrire_B(self, membre, cle):
     if cle in self.reponses: return self.reponses[cle]
     n = len(self.inscrits)
     yield
     r = "complet"
     if n < CAPACITE: self.inscrits.append((membre, cle)); r = "confirmé"
     self.reponses[cle] = r; return r
 # --- conception C : B + écrivain unique (vérifier-puis-agir ATOMIQUE, sans po
int d'entrelacement) + réplication avant confirmation ---
 def inscrire_C(self, membre, cle):
     if cle in self.reponses: return self.reponses[cle]
     if any(m == membre for m, _ in self.inscrits): r = "déjà inscrit"
     elif len(self.inscrits) < CAPACITE:
         self.inscrits.append((membre, cle)); self.replique.append((membre, cle
)); r = "confirmé"    # répliqué AVANT de répondre
     else: r = "complet"
     self.reponses[cle] = r
     return r
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        yield                                                    # (fait de
   cette méthode un générateur)

def client(service, reseau, membre, cle, journal, essais=4):
    # Un client qui relance avec la même clé tant qu'il n'a pas de réponse.
    operation = getattr(service, f"inscrire_{service.c}")
    for _ in range(essais):
        if reseau.rng.random() < reseau.pr: yield; continue
          # requête perdue : délai puis relance
        copies = 2 if reseau.rng.random() < reseau.pd else 1
          # doublon réseau : la requête arrive 2 fois
        resultat = None
        for _ in range(copies):
            resultat = yield from operation(membre, cle)
        if reseau.rng.random() < reseau.pp: yield; continue
          # réponse perdue : le client ne sait pas
        journal.append((membre, resultat)); return
    journal.append((membre, "inconnu"))
```

Exemple pas à pas (suite)

```
def executer(conception, graine, nb_membres=12, nb_doubles_clics=4):
    rng = random.Random(graine); reseau = Reseau(rng); service = Service(conceptio
   n); journal = []
    taches = []
    for i in range(nb_membres):
        taches.append(client(service, reseau, f"m{i}", f"cle-m{i}", journal))
    for i in range(nb_doubles_clics):
            # des utilisateurs cliquent deux fois : NOUVELLE intention (nouvelle c
   lé) pour la même personne
        taches.append(client(service, reseau, f"m{i}", f"cle-m{i}-bis", journal))
    while taches:
            # ordonnanceur : entrelacement aléatoire
        t = rng.choice(taches)
        try: next(t)
        except StopIteration: taches.remove(t)
    return service, journal

def verifier(service, journal):
    violations = []
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    membres = [m for m, _ in service.inscrits]
    if len(service.inscrits) > CAPACITE: violations.append("surbooking")
    if len(membres) != len(set(membres)): violations.append("doublon")
    confirmes = {m for m, r in journal if r == "confirmé"}
    if not confirmes <= set(membres): violations.append("inscription confirmée per
   due")
    return violations

print("conception | exécutions avec violation (sur 500) | détail")
bilan = {}
for conception in "ABC":
    compte = {"surbooking": 0, "doublon": 0, "inscription confirmée perdue": 0}; f
   autives = 0
    for graine in range(500):
        service, journal = executer(conception, graine)
        v = verifier(service, journal)
        fautives += bool(v)
        for x in v: compte[x] += 1
    bilan[conception] = (fautives, compte)
```

Exemple pas à pas (suite)

```
    print(f"     {conception}     |             {fautives:4d}               | {c
   ompte}")

assert bilan["A"][0] > 100 and bilan["A"][1]["doublon"] > 0 and bilan["A"][1]["sur
   booking"] > 0     # naïve : plusieurs invariants violés
assert bilan["B"][1]["doublon"] < bilan["A"][1]["doublon"] and bilan["B"][1]["surb
   ooking"] > 0        # la clé règle les doublons de requête mais pas le surbooki
   ng
assert bilan["C"][0] == 0
                       # écrivain unique + clé + réplication : aucun invariant vi
   olé sur 500 exécutions
```

Lecture :

Exemple pas à pas (suite)

Conception	Mécanismes	Résultat sur 500 exécutions aléatoires
A (naïve)	aucun	surbooking **et** doublons (relances, doublons réseau, doubles clics) dans presque toutes les exécutions
B	+ clé d'idempotence	les doublons dus aux **relances et au réseau** disparaissent, mais pas les **doubles clics** (nouvelle intention) ni le **surbooking** : deux inscriptions lisent la même place libre
C	+ écrivain unique (vérifier-puis-agir **atomique**, règle « déjà inscrit ») + clé + réplication avant confirmation	**zéro violation** sur 500 exécutions

Exemple pas à pas (suite)

Le banc d'essai **trouve** des défauts de conception que des tests unitaires ne verraient jamais (entrelacements rares). Chaque mécanisme ajouté répond à une **menace précise**, et le test le **prouve** en le supprimant : la conception B montre ce que la clé apporte ET ce qu'elle ne suffit pas à garantir. Un doublon de personne (même membre, deux clés différentes) n'est évité que par la règle métier « déjà inscrit », ajoutée en C.

À toi, avec aide

Rédige l'**ADR** (registre de décision d'architecture) de la décision « partitionner les inscriptions par cours avec un leader par partition, réplication à la majorité » pour l'événement de HashBox :

1. **Contexte** : les exigences (invariants, volumes) ;
2. **Options** considérées : (a) une base unique, (b) partitionnement par cours avec leader, (c) quorum sans leader (R + W > N) ;
3. **Décision** et **raisons** (une phrase par invariant) ;
4. **Conséquences** positives et négatives, **risques** et comment les surveiller ;
5. **Tests** qui valident la décision.

Écris-le sous forme de dictionnaire Python, puis une petite fonction qui vérifie que l'ADR est complet (toutes les sections non vides, chaque invariant relié à un mécanisme et à un test).

À toi, avec aide · réponse

Indices progressifs

Indice 1

Utilise les invariants de la mission : pas de surbooking, pas de doublon, pas de perte.

Indice 2

Option (c) : un quorum seul ne rend pas « vérifier puis agir » atomique.

Indice 3

Conséquence négative : un cours très populaire = un point chaud (mission 6).

À toi, avec aide · réponse

Réponse

```
ADR = {
    "titre": "ADR-007 : partitionner les inscriptions par cours avec un leader par
    partition",
    "contexte": "5 000 personnes, 40 places par cours, clics simultanés et doubles
    ; invariants : jamais > 40 inscrits, jamais de doublon, jamais de perte d'une
   inscription confirmée.",
    "options": {"a": "base unique : simple mais point de défaillance unique et déb
   it limité",
                "b": "partition par cours, un leader par partition, réplication à
   la majorité",
                "c": "quorum sans leader (R + W > N) : bonne disponibilité mais ne
    rend pas atomique « vérifier puis agir »"},
    "decision": "option b",
    "raisons": {"pas de surbooking": "un seul écrivain par cours rend le contrôle
   de capacité atomique",
                "pas de doublon": "clé d'idempotence par intention + règle « déjà
   inscrit »",
                "pas de perte": "confirmation après réplication à la majorité ; él
   ection à la majorité"},
```

À toi, avec aide · réponse

Réponse (suite)

```
    "consequences": {"positives": ["invariants simples à garantir", "partitions in
   dépendantes : parallélisme"],
                    "negatives": ["un cours très demandé = point chaud", "indispo
   nibilité de la partition minoritaire pendant une coupure"],
                    "risques": ["cerveau fendu après basculement : atténué par je
   tons de barrière", "retard de réplication : surveillé par une alerte"]},
    "tests": {"pas de surbooking": "banc d'essai 500 graines, invariant capacité",
    "pas de doublon": "relances avec clés, double-clic",
              "pas de perte": "panne du leader après confirmation"},
}

def verifier_adr(adr):
    manques = [k for k in ("titre", "contexte", "options", "decision", "raisons",
   "consequences", "tests") if not adr.get(k)]
    for inv in adr["raisons"]:
        if inv not in adr["tests"]: manques.append(f"invariant sans test : {inv}")
    if not adr["consequences"].get("negatives"): manques.append("aucune conséquenc
   e négative : suspect")
    return manques
```

À toi, avec aide · réponse

Réponse (suite)

```
assert verifier_adr(ADR) == []
assert verifier_adr(dict(ADR, tests={})) != []
print("ADR complet : chaque invariant a sa raison et son test ; des conséquences n
   égatives sont assumées")
```

Un ADR utile **assume des inconvénients** (point chaud, indisponibilité minoritaire). Une décision sans conséquence négative n'a probablement pas été examinée honnêtement.

À toi, en solo

Étendre le banc d'essai : panne du leader et basculement. Ajoute à la conception C de l'exemple :

1. deux réplicas (3 copies) et un **leader** ; l'inscription est confirmée au client **après** écriture sur au moins 2 copies ;
2. une panne aléatoire du leader pendant l'exécution : un **basculement** élit le réplica le plus à jour (parmi les copies disponibles, au moins 2 sur 3) ;
3. une variante **dangereuse** : confirmation après écriture sur le leader seul (réplication asynchrone) ;
4. sur 500 graines avec panne du leader, vérifie l'invariant « toute inscription confirmée survit à la panne » : la variante dangereuse doit en perdre, la variante à quorum non ;
5. explique pourquoi, dans la variante à quorum, les **clés d'idempotence** doivent elles aussi être répliquées si l'on veut que la relance d'un client (après la perte d'une réponse) obtienne la réponse d'origine (« confirmé ») plutôt que « déjà inscrit ».

À toi, en solo · réponse

Pistes de correction

```
import random

CAP = 5
class Groupe:
    def __init__(self, replication):
        self.copies = [[], [], []]                # inscrits par copie (0 = lead
   er au départ)
        self.leader, self.vivants, self.replication = 0, {0, 1, 2}, replication
        self.cles = [{}, {}, {}]                   # réponses mémorisées par cop
   ie
    def panne_leader(self):
        self.vivants.discard(self.leader)
        self.leader = max(sorted(self.vivants), key=lambda c: len(self.copies[c]))
         # basculement : la copie la plus à jour
    def inscrire(self, membre, cle):
        L = self.leader
        if cle in self.cles[L]: return self.cles[L][cle]
        if membre in self.copies[L]: r = "déjà inscrit"
        elif len(self.copies[L]) >= CAP: r = "complet"
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        else:
            self.copies[L].append(membre)
            if self.replication == "quorum":
                autre = next(c for c in sorted(self.vivants) if c != L)
                self.copies[autre].append(membre)                          # écrit
    sur une 2e copie AVANT de confirmer au client
            r = "confirmé"
        self.cles[L][cle] = r
        return r

def execution(replication, graine):
    rng = random.Random(graine); g = Groupe(replication)
    membres, confirmes = [f"m{i}" for i in range(CAP)], set()
    panne_a = rng.randrange(1, CAP + 1)             # le leader tombe avant la req
   uête n° panne_a (ou après la dernière)
    for i, m in enumerate(membres):
        if i == panne_a: g.panne_leader()
        if g.inscrire(m, f"cle-{m}") == "confirmé": confirmes.add(m)
    if panne_a == CAP: g.panne_leader()
```

Pistes de correction (suite)

```
    return len(confirmes - set(g.copies[g.leader]))                # inscription
   s confirmées au client mais absentes après le basculement

for nom, repl in (("asynchrone (leader seul)", "async"), ("quorum (2 copies)", "qu
   orum")):
    pertes = sum(1 for g in range(500) if execution(repl, g) > 0)
    print(f"{nom:26s} → exécutions avec inscription confirmée PERDUE : {pertes}/50
   0")
assert sum(1 for g in range(500) if execution("async", g) > 0) > 300
assert sum(1 for g in range(500) if execution("quorum", g) > 0) == 0
```

La confirmation **après réplication** à une majorité est ce qui protège les inscriptions confirmées d'une panne du leader. En mode asynchrone, une inscription confirmée au client mais pas encore répliquée **disparaît** avec le leader : c'est la perte que le banc d'essai révèle. Répliquer aussi les **clés d'idempotence** (réponses mémorisées) est nécessaire pour qu'une relance après basculement obtienne la réponse d'origine (« confirmé ») plutôt qu'un « déjà inscrit » déroutant.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Énoncer des exigences sans invariants vérifiables	On ne peut pas tester ce qu'on ne sait pas dire	Invariants de sûreté explicites, mesurables
Tester seulement le cas heureux	Les bugs de concurrence et de panne restent invisibles	Banc d'essai avec fautes aléatoires et graines
Choisir un mécanisme sans le relier à une menace	Complexité inutile ou trous de sécurité	Table menace → mécanisme → test
Documenter sans assumer les inconvénients	Décisions non réexaminées	ADR avec conséquences négatives et risques

Mini-boss

Projet : architecture d'un mini-service distribué avec preuves par simulation. Choisis un service (inscriptions en direct de HashBox, ou une alternative : réservation de ressources, compteur de stock, file de tâches) et livre :

1. **exigences** et **invariants** de sûreté (au moins 4) + objectifs de vivacité/disponibilité ;
2. **architecture** : diagramme Mermaid (composants, flux, partitions, réplicas), **contrats** (API, événements, erreurs, idempotence), choix de partitionnement et réplication justifiés ;
3. **table menace → mécanisme → test** (au moins 8 lignes) ;
4. **simulateur déterministe** : processus, réseau avec pertes/duplications/retards/réordonnancements, pannes et redémarrages de nœuds, clients qui relancent ;
5. **banc d'essai** : au moins 1000 graines, invariants vérifiés, rapport avec nombre de violations par invariant ;
6. **deux conceptions dégradées** (en supprimant un mécanisme à la fois) pour prouver que chaque mécanisme est nécessaire ;

Mini-boss

Mini-boss (suite)

7. **minimisation** : pour une graine en échec, réduis le scénario à moins de 15 événements et décris la cause ;
8. **ADR** (au moins 3 décisions), **runbook** de 3 pannes, **plan d'exercice de panne** ;
9. **rétrospective** (15 lignes) : quel mécanisme a le plus apporté, ce que la simulation n'a pas couvert (réalisme du réseau, horloges, disque), quelles garanties tu ne donnerais **pas** en production sans test réel.

Mini-boss · réponse

Pistes de correction

Un bon projet : exigences et invariants précis; diagramme et contrats; table menace → mécanisme → test avec au moins une ligne par mission du module (horloges/causalité, RPC et délais, retries et budget, idempotence, réplication et quorum, partitionnement, consensus ou leader, messages et outbox, pannes et compensations); simulateur avec graine; rapport de violations par invariant pour la version complète (0) et les versions dégradées (>0, chaque mécanisme retiré fait apparaître une violation distincte); minimisation par suppression itérative d'événements (delta debugging); ADR sans complaisance; rétrospective honnête sur les limites de la simulation.

Quiz éclair

Quiz éclair

1. Un invariant de sûreté est :

A. une propriété qui doit être vraie dans toute exécution, par exemple « jamais plus de N inscrits »
B. un objectif de vitesse
C. une fonctionnalité optionnelle

2. Le banc d'essai à graines fixées permet :

A. de supprimer toutes les pannes
B. de rejouer à l'identique une exécution qui a violé un invariant
C. d'éviter d'écrire des tests

3. Pour garantir « pas de surbooking » malgré des clics concurrents, il faut notamment :

Quiz éclair (suite)

A. plus de serveurs
B. une clé d'idempotence seule
C. un contrôle de capacité atomique, par exemple un écrivain unique par cours

4. 🔁 *Révision mission 09* : Un jeton de barrière (fencing token) empêche :

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

5. 🔁 *Révision mission 09* : Dans une saga, les compensations s'exécutent :

A. dans le même ordre, une seule fois
B. dans l'ordre inverse des étapes déjà faites, et doivent être idempotentes

Quiz éclair

Quiz éclair (suite)

C. sans ordre

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quels invariants de ton système pourraient être violés par une panne rare, sans qu'aucun test ne l'ait jamais essayé ?
2. Quel mécanisme de ce module ajouterais-tu en premier à un système que tu connais ?

À retenir

1. Exigences → invariants de sûreté → mécanismes → scénarios de panne → vérification.
2. Table menace → mécanisme → test ; chaque mécanisme répond à une menace.
3. Simulation déterministe avec fautes aléatoires : trouve les entrelacements rares.
4. Supprimer un mécanisme et montrer la violation prouve sa nécessité.
5. Documenter : diagrammes, contrats, ADR avec inconvénients assumés, runbooks.

Félicitations

Félicitations

Tu as terminé **HEF-140 — Systèmes distribués**. Tu sais raisonner avec des pannes partielles et sans horloge commune, concevoir des échanges fiables (délais, retries, idempotence), répliquer et partitionner des données, énoncer CAP sans caricature, comprendre un consensus de type Raft, utiliser messages et événements, et vérifier par simulation qu'un service tient ses invariants sous pannes.
→ Prochain module : **HEF-150 — Ingénierie produit**

MISSION ACCOMPLIE

+120 XP

Badge débloqué : Architecte du distribué

Fin du module HEF-140. Bravo !

