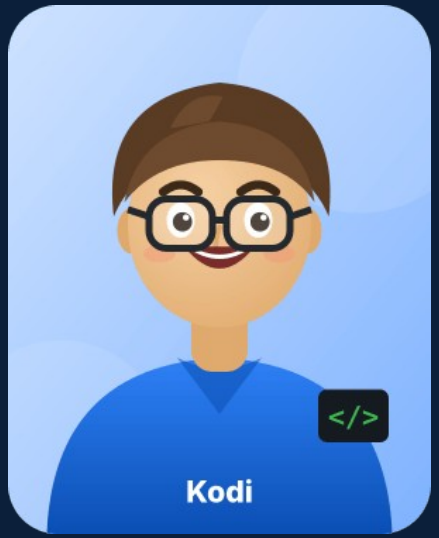
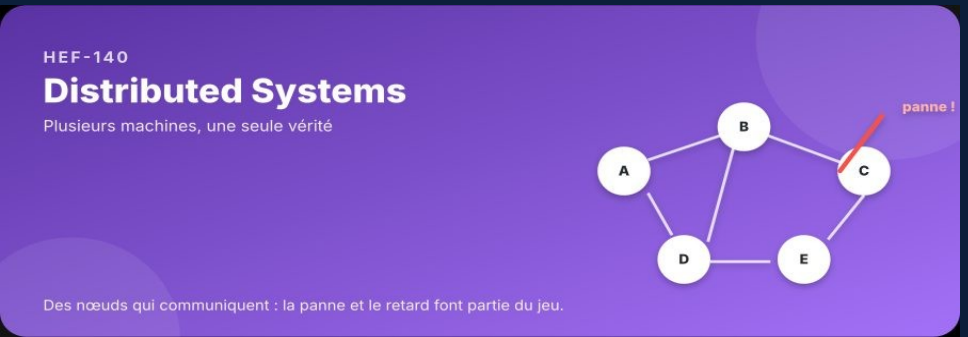
HEF-140 · MISSION 01 / 10

Modèles et contraintes distribuées

Dans un système distribué, il n'y a pas d'horloge commune, pas de mémoire partagée, et une partie du système peut tomber sans que le reste le sache.





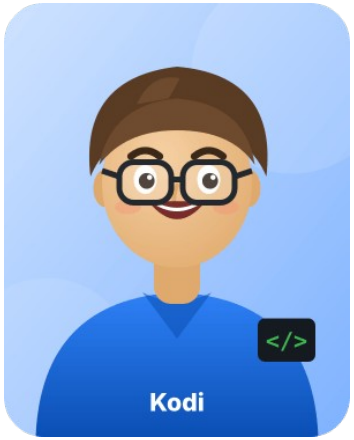
Badge : Explorateur du distribué

100 min · avancé

L'histoire

HashBox ouvre un second centre de données à Montréal pour se rapprocher de ses utilisateurs canadiens. Deux jours plus tard, un formateur de Lyon et une formatrice de Montréal modifient le même cours à quelques secondes d'écart. Résultat : la modification de l'un écrase celle de l'autre, et le journal des événements montre l'inscription de Léo **avant** la création du cours, ce qui est impossible.

« Tu viens de rencontrer les trois difficultés fondamentales, dit Sam. Les machines ne s'accordent pas sur l'heure. Les messages se perdent ou arrivent en retard. Et quand une réponse ne vient pas, tu ne sais pas si l'autre est tombé, lent, ou si c'est le réseau. Il faut raisonner autrement. »



L'histoire (suite)

Tu apprends les propriétés d'un système distribué, ses modèles de pannes, pourquoi aucune horloge ne suffit, et comment les horloges logiques de Lamport et les horloges vectorielles ordonnent les événements.

Ta mission

À la fin, je serai capable de…

1. énumérer les difficultés propres au distribué (pannes partielles, latence, absence d'horloge globale)
2. décrire les modèles de pannes (arrêt, omission, byzantine) et de synchronie
3. expliquer le problème des deux généraux et ses conséquences
4. implémenter des horloges de Lamport et vectorielles
5. décider si deux événements sont causalement liés ou concurrents

Défi d'échauffement

Défi d'échauffement

Deux amis décident de se retrouver au café, chacun dans une ville différente, uniquement par SMS qui peuvent se perdre. Ada envoie « on se retrouve à 18 h ? », Bob répond « ok ». Ada est-elle sûre que Bob a reçu son message ? Bob est-il sûr qu'Ada a reçu son « ok » ? Combien de SMS faut-il pour que les deux soient *certains* ? Existe-t-il un nombre qui fonctionne ?

Le concept

Qu'est-ce qu'un système distribué ?

Un ensemble de **machines indépendantes** qui coopèrent en échangeant des **messages** pour offrir un service unique. Raisons de le faire : **capacité** (partager la charge), **disponibilité** (survivre aux pannes), **latence** (se rapprocher des utilisateurs), **organisation** (équipes indépendantes).
Le prix : un ensemble de difficultés qui n'existent pas sur une seule machine.

Les huit erreurs classiques (« fallacies »)

Les développeurs croient souvent que :

1. le réseau est fiable ;
2. la latence est nulle ;

Le concept

Les huit erreurs classiques (« fallacies ») (suite)

3. la bande passante est infinie ;
4. le réseau est sûr ;
5. la topologie ne change pas ;
6. il y a un seul administrateur ;
7. le coût de transport est nul ;
8. le réseau est homogène.

Tout est faux. Un système distribué correct commence par refuser ces hypothèses.

Trois difficultés fondamentales

Le concept

Trois difficultés fondamentales (suite)

Difficulté	Conséquence
Pannes partielles	une partie du système tombe pendant que le reste fonctionne ; on ne sait pas toujours laquelle
Asynchronie	les messages ont un délai variable, non borné ; on ne peut pas distinguer « **lent** » de « **mort** »
Pas d'horloge ni de mémoire communes	pas d'ordre global évident des événements

La question « le serveur est-il tombé ? » n'a **pas de réponse sûre** : un délai dépassé signifie « je n'ai pas eu de réponse », pas « il est mort ».

Les modèles de pannes

Modèle	Comportement du nœud défaillant
Arrêt sur panne (*crash-stop*)	s'arrête définitivement

Le concept

Les modèles de pannes (suite)

Modèle	Comportement du nœud défaillant
Reprise après panne (*crash-recovery*)	s'arrête puis redémarre (peut avoir perdu sa mémoire)
Omission	perd des messages (envoi ou réception)
Partition réseau	des groupes de nœuds ne peuvent plus communiquer entre eux
Byzantine	comportement arbitraire, y compris malveillant

Plus le modèle de panne est sévère, plus les algorithmes sont coûteux. La plupart des systèmes d'entreprise supposent *crash-recovery* + partitions.

Les modèles de synchronie

Modèle	Hypothèse
Synchrone	délais bornés connus, horloges précises

Le concept

Les modèles de synchronie (suite)

Modèle	Hypothèse
Asynchrone	aucune borne sur les délais
Partiellement synchrone	des bornes existent mais sont inconnues, ou valables « finalement »

Le monde réel est **partiellement synchrone** : on conçoit des algorithmes **sûrs** même en asynchrone (jamais de mauvais résultat), et **vivants** quand le réseau se comporte bien.

Le problème des deux généraux

Deux armées doivent attaquer simultanément pour gagner, en communiquant par messagers qui peuvent être interceptés. **Aucun protocole fini ne garantit l'accord** : celui qui envoie le dernier message ne sait jamais s'il est arrivé. On peut réduire la probabilité de désaccord, jamais l'annuler : c'est l'impossibilité fondamentale derrière les confirmations, les délais, les transactions distribuées.

Le concept

Pas d'horloge commune

Les horloges physiques **dérivent** (millisecondes par jour) et ne sont synchronisées (NTP) qu'à quelques millisecondes près, parfois bien plus. Trier des événements de machines différentes par heure physique donne des ordres **faux**.

La relation « s'est produit avant » (Lamport)

On note $a \to b$ (a précède causalement b) si :

1. a et b sont sur le même processus et a vient avant b ;
2. a est l'envoi d'un message et b sa réception ;
3. transitivité : $a \to c$ et $c \to b \Longrightarrow a \to b$.

Le concept

La relation « s'est produit avant » (Lamport) (suite)

Si ni $a \to b$ ni $b \to a$, les événements sont **concurrents** ($a \parallel b$) : aucun ne peut avoir influencé l'autre.

Horloges de Lamport (logiques)

Chaque processus tient un compteur L.

- événement local ou envoi : $L \leftarrow L + 1$ (le message porte L) ;
- réception d'un message portant Lm : $L \leftarrow \max(L, Lm) + 1$.

Propriété : $a \to b \Longrightarrow L(a) < L(b)$. **Mais la réciproque est fausse** : $L(a) < L(b)$ ne prouve pas que $a \to b$ (ils peuvent être concurrents).

Horloges vectorielles

Chaque processus tient un **vecteur** de compteurs (un par processus).

Le concept

Horloges vectorielles (suite)

- événement local ou envoi : `V[i] += 1` ;
- réception de `Vm` : `V ← max terme à terme(V, Vm)` puis `V[i] += 1`.

Comparaison :

- `Va ≤ Vb` partout et `Va ≠ Vb` ⟹ `a → b` ;
- ni l'un ni l'autre ⟹ **concurrents**.

Les horloges vectorielles **caractérisent exactement** la causalité (iff). Elles servent à détecter les **conflits** d'écriture concurrente (Dynamo, Riak).

Idées-forces

- Concevoir pour la panne : **timeouts**, **reprises**, **idempotence** (mission 4).

Le concept

Idées-forces (suite)

- Ne jamais supposer un ordre global : utiliser la causalité, un journal ordonné, ou un consensus (mission 7).
- **Mesurer** : latence, taux d'erreur, retries.

🧠 ***Astuce*** *: devant un incident distribué, demande-toi toujours : que sait réellement chaque nœud à cet instant ? Souvent, personne ne connaît l'état global.*

Exemple pas à pas

Problème : (1) simuler le problème des deux généraux, (2) implémenter les horloges de Lamport et vectorielles sur un scénario à trois processus, (3) montrer qu'une horloge physique décalée donne un ordre faux alors que l'horloge vectorielle capture la causalité.

```
import random

# ---------------- 1. les deux généraux ----------------
def deux_generaux(p_perte, k, essais=100000, graine=1):
    # protocole à k messages alternés A→B, B→A, ... ; le dernier expéditeur attaqu
   e sans attendre de confirmation
    rng = random.Random(graine)
    les_deux = desaccord = personne = 0
    for _ in range(essais):
        livres = 0
        for i in range(k):
            if rng.random() < p_perte: break                              # ce messa
   ge est perdu : la chaîne s'arrête
            livres += 1
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        if livres == k:       les_deux += 1                                  # tous reç
   us : les deux attaquent
        elif livres == k - 1: desaccord += 1                                 # seul le
   dernier est perdu : l'expéditeur attaque seul
        else:                 personne += 1                                  # une pert
   e plus tôt : personne n'attaque (cohérent)
    return les_deux / essais, desaccord / essais, personne / essais

p = 0.2
print("k messages | les deux attaquent | DÉSACCORD | personne n'attaque")
for k in (1, 2, 4, 8, 16):
    a, d, n = deux_generaux(p, k)
    theorique_d = p * (1 - p) ** (k - 1)
    print(f"   {k:2d}      |       {a:6.1%}       |  {d:6.2%}   |      {n:6.1%}
   (désaccord théorique {theorique_d:6.2%})")
    assert abs(d - theorique_d) < 0.01
assert deux_generaux(p, 16)[1] > 0.0                                           # jamais
    zéro : l'accord certain est impossible
assert deux_generaux(p, 16)[0] < deux_generaux(p, 2)[0]                        # et plu
```

Exemple pas à pas

Exemple pas à pas (suite)

```
   s on exige de confirmations, moins on attaque

# ---------------- 2. horloges de Lamport et vectorielles ----------------
class Processus:
    def __init__(self, nom, noms):
        self.nom, self.i = nom, noms.index(nom)
        self.lamport, self.vecteur, self.journal = 0, [0] * len(noms), []
    def evenement(self, description, heure_physique=None):
        self.lamport += 1; self.vecteur[self.i] += 1
        self.journal.append((description, self.lamport, tuple(self.vecteur), heure
   _physique)); return self
    def envoyer(self, description, heure_physique=None):
        self.evenement(description, heure_physique)
        return {"lamport": self.lamport, "vecteur": tuple(self.vecteur)}
    def recevoir(self, message, description, heure_physique=None):
        self.lamport = max(self.lamport, message["lamport"])
        self.vecteur = [max(a, b) for a, b in zip(self.vecteur, message["vecteur"]
   )]
        self.evenement(description, heure_physique)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
def compare(va, vb):
    if va == vb: return "égaux"
    if all(a <= b for a, b in zip(va, vb)): return "avant"
    if all(a >= b for a, b in zip(va, vb)): return "après"
    return "concurrents"

NOMS = ["Lyon", "Montreal", "Paris"]
lyon, mtl, paris = (Processus(n, NOMS) for n in NOMS)
# Lyon crée le cours (horloge physique : 10:00:05), l'envoie à Montréal (qui a 8 s
    de RETARD : 10:00:02 à la réception)
m1 = lyon.envoyer("création du cours", "10:00:05")
mtl.recevoir(m1, "Montréal reçoit le cours", "10:00:02")                # heure p
   hysique antérieure à la création : horloge en retard
m2 = mtl.envoyer("inscription de Léo", "10:00:03")
paris.evenement("Paris modifie le prix", "10:00:04")                     # indépe
   ndant de tout le reste
paris.recevoir(m2, "Paris reçoit l'inscription de Léo", "10:00:06")
```

Exemple pas à pas

Exemple pas à pas (suite)

```
evts = {d: (l, v, h) for p in (lyon, mtl, paris) for d, l, v, h in p.journal}
creation, inscription, prix = evts["création du cours"], evts["inscription de Léo"
   ], evts["Paris modifie le prix"]
print("heure physique :", "création", creation[2], "| inscription", inscription[2]
   , "→ l'inscription semble antérieure à la création !")
assert inscription[2] < creation[2]                                    # l'horl
   oge physique donne un ordre FAUX
assert compare(creation[1], inscription[1]) == "avant"                 # l'horl
   oge vectorielle l'ordonne correctement
assert creation[0] < inscription[0]                                    # Lampor
   t aussi (a → b ⇒ L(a) < L(b))
assert compare(prix[1], inscription[1]) == "concurrents"               # le cha
   ngement de prix est indépendant de l'inscription
# Lamport ne distingue pas « concurrent » de « avant » : la réciproque est fausse
print(f"Lamport : prix = {prix[0]}, inscription = {inscription[0]} → ordonnés, alo
   rs qu'ils sont concurrents (vecteurs : {prix[1]} et {inscription[1]})")
assert prix[0] < inscription[0] and compare(prix[1], inscription[1]) == "concurren
   ts"
for d, l, v, h in sorted((e for p in (lyon, mtl, paris) for e in p.journal), key=l
```

Exemple pas à pas (suite)

```
    ambda e: e[1]):
     print(f"  L={l}  V={v}  {d}")
```

Lecture :

1. **Deux généraux** : quel que soit `k`, le désaccord n'est jamais nul : il vaut `p(1 − p)^(k−1)`. Exiger plus de confirmations **réduit** le désaccord mais **réduit aussi** la probabilité d'attaquer. On ne gagne jamais la certitude : on **choisit** un compromis entre sûreté et activité.
2. **Horloge physique** : Montréal a huit secondes de retard ; trier par heure physique place l'inscription de Léo **avant** la création du cours. C'est faux.
3. **Lamport** respecte la causalité (`création → inscription` implique `L(création) < L(inscription)`), mais il ordonne aussi des événements **concurrents** (`prix` et `inscription`) sans le dire.
4. **Vecteurs** : ils distinguent « avant » de « concurrents » : le changement de prix de Paris est concurrent à l'inscription. C'est exactement l'information nécessaire pour détecter un **conflit**.

À toi, avec aide

À toi, avec aide

Trois processus A, B, C ont produit ces événements avec leurs horloges vectorielles `[A, B, C]` :

Événement	Vecteur
e1 (A)	[1, 0, 0]
e2 (A, envoi à B)	[2, 0, 0]
e3 (B, réception de e2)	[2, 1, 0]
e4 (C)	[0, 0, 1]
e5 (B, envoi à C)	[2, 2, 0]
e6 (C, réception de e5)	[2, 2, 2]

Écris `relation(x, y)` et affiche la relation pour **toutes** les paires. Combien de paires sont concurrentes ? Lesquelles ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Utilise la fonction `compare` de l'exemple.

Indice 2

Utilise `itertools.combinations` pour les paires.

Indice 3

e4 n'a aucune relation causale avec e1, e2, e3, e5 (sauf par e6).

À toi, avec aide · réponse

Réponse

```
from itertools import combinations
V = {"e1": (1, 0, 0), "e2": (2, 0, 0), "e3": (2, 1, 0), "e4": (0, 0, 1), "e5": (2,
    2, 0), "e6": (2, 2, 2)}

def compare(va, vb):
    if va == vb: return "égaux"
    if all(a <= b for a, b in zip(va, vb)): return "avant"
    if all(a >= b for a, b in zip(va, vb)): return "après"
    return "concurrents"

concurrentes = [(x, y) for x, y in combinations(V, 2) if compare(V[x], V[y]) == "c
   oncurrents"]
for x, y in combinations(V, 2):
    print(f"{x} {compare(V[x], V[y]):12s} {y}")
print("paires concurrentes :", concurrentes)
assert concurrentes == [("e1", "e4"), ("e2", "e4"), ("e3", "e4"), ("e4", "e5")]
```

Réponse (suite)

Quatre paires sont concurrentes : **e4 avec e1, e2, e3, e5**. C'est cohérent : le processus C a fait quelque chose d'indépendant (e4) avant de recevoir le message de B (e5). L'événement e6 est postérieur à tous les autres car il a reçu la causalité de A et de B.

À toi, en solo

Détecteur de conflits avec versions vectorielles. Implémente un petit magasin clé-valeur répliqué sur 3 nœuds où chaque valeur porte un **vecteur de version** (comme Dynamo) :

1. `ecrire(noeud, cle, valeur, contexte)` : l'écriture incrémente le compteur du nœud dans le vecteur du contexte lu ;
2. chaque nœud garde, pour une clé, **l'ensemble des versions non dominées** (frères) ;
3. `fusionner(noeud_a, noeud_b)` : échange de versions (anti-entropie) : une version dominée (strictement avant) est supprimée, deux versions concurrentes sont **toutes deux conservées** ;
4. `lire(noeud, cle)` renvoie les frères et un contexte (le maximum des vecteurs) ;
5. scénario : deux clients lisent la même version puis écrivent chacun de leur côté (sur deux nœuds différents) pendant une partition ; après fusion, la lecture renvoie **deux frères** ; un client résout le conflit en écrivant avec le contexte fusionné : un seul reste.

À toi, en solo · réponse

Pistes de correction

```
def domine(va, vb):                                  # va ≤ vb partout et différen
   t
    return all(a <= b for a, b in zip(va, vb)) and va != vb

class Noeud:
    def __init__(self, i, n=3):
        self.i, self.n, self.donnees = i, n, {}      # cle -> liste de (valeur, ve
   cteur)
    def ecrire(self, cle, valeur, contexte=None):
        v = list(contexte or (0,) * self.n); v[self.i] += 1; v = tuple(v)
        versions = [(val, vec) for val, vec in self.donnees.get(cle, []) if not do
   mine(vec, v)]
        self.donnees[cle] = versions + [(valeur, v)]
        return v
    def lire(self, cle):
        versions = self.donnees.get(cle, [])
        contexte = tuple(max(c) for c in zip(*[v for _, v in versions])) if versio
   ns else (0,) * self.n
        return sorted(val for val, _ in versions), contexte
```

À toi, en solo · réponse

Pistes de correction (suite)

```
def fusionner(a, b):
    for cle in set(a.donnees) | set(b.donnees):
        toutes = list({(val, vec) for val, vec in a.donnees.get(cle, []) + b.donne
   es.get(cle, [])})
        gardees = [(val, vec) for val, vec in toutes if not any(domine(vec, autre)
    for _, autre in toutes)]
        a.donnees[cle] = gardees; b.donnees[cle] = list(gardees)

n0, n1, n2 = Noeud(0), Noeud(1), Noeud(2)
n0.ecrire("cours:42", "prix=40"); fusionner(n0, n1); fusionner(n0, n2)          #
   tous voient « prix=40 »
valeur, contexte = n1.lire("cours:42"); assert valeur == ["prix=40"]
# partition : deux clients écrivent depuis le MÊME contexte sur deux nœuds différe
   nts
n0.ecrire("cours:42", "prix=45", contexte)
n2.ecrire("cours:42", "prix=50", contexte)
fusionner(n0, n2)                                                              #
    la partition est réparée
```

À toi, en solo · réponse

Pistes de correction (suite)

```
frères, ctx = n0.lire("cours:42")
print("après fusion :", frères)
assert frères == ["prix=45", "prix=50"]                                    #
    CONFLIT : deux versions concurrentes
n1_apres = n0.ecrire("cours:42", "prix=47", ctx)                           #
    résolution par le client, avec le contexte fusionné
frères, _ = n0.lire("cours:42")
print("après résolution :", frères, n1_apres)
assert frères == ["prix=47"]
```

Le système ne **choisit pas** entre 45 et 50 (contrairement à « le dernier gagne », qui perdrait silencieusement une écriture) : il **expose le conflit**. C'est au client — ou à une règle métier, ou à un type de données fusionnable (mission 5) — de le résoudre.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Trier des événements par heure physique entre machines	Ordre faux, effets de causalité inversés	Horloges logiques/vectorielles, ou journal ordonné par un consensus
Supposer qu'un délai dépassé signifie une panne	On ne sait pas si c'est lent, mort ou le réseau	Raisonner en incertitude : retries idempotents, détecteurs de pannes
Appliquer « le dernier gagne » sans réfléchir	Écriture perdue silencieusement	Détecter les conflits (vecteurs) ou utiliser des types fusionnables
Croire que le réseau est fiable	Pannes en production	Concevoir pour la perte, la duplication, le retard, le réordonnancement

Mini-boss

Mini-boss

Journal causal de HashBox. Trois services (inscriptions, paiements, notifications) produisent des événements horodatés avec leurs horloges physiques (décalées de plusieurs secondes) et échangent des messages. Génère un scénario de 30 événements (graine fixée) avec des envois/réceptions.

1. équipe chaque processus d'horloges de Lamport et vectorielles ;
2. écris `ordre_causal(evenements)` qui produit **un** ordre total compatible avec la causalité (tri topologique) ;
3. montre que le tri par heure physique viole au moins une relation de causalité, mais pas celui de ton algorithme ;
4. écris `paires_concurrentes(evenements)` et liste celles qui touchent la même ressource (conflits potentiels) ;
5. vérifie par un test aléatoire (100 graines) que ton ordre respecte `a → b ⇒ a avant b` et que Lamport respecte la même propriété mais que la réciproque est fausse dans au moins un cas ;

Mini-boss (suite)

6. écris 5 lignes sur ce que ton système ne peut pas savoir (ex. : l'ordre « réel » des événements concurrents).

Mini-boss · réponse

Pistes de correction

Un bon travail : classes `Processus` avec vecteur et Lamport, génération aléatoire d'actions (local, envoi, réception d'un message en attente), tri topologique (par exemple tri par somme du vecteur puis départage par nom — valide car `a → b` implique somme(Va) < somme(Vb)), détection des violations du tri physique, test aléatoire sur 100 graines, liste des conflits (paires concurrentes sur la même clé). Le point final : l'ordre entre événements concurrents est arbitraire et non observable — toute application doit être correcte quel que soit cet ordre.

Quiz éclair

Quiz éclair

1. Dans un système distribué asynchrone, un délai dépassé signifie :

A. qu'on n'a pas reçu de réponse, sans savoir si l'autre est lent, mort ou injoignable
B. que l'autre machine est tombée
C. que le message est arrivé

2. $a \rightarrow b$ (a s'est produit avant b) signifie :

A. que l'horloge physique de a est plus petite
B. que a peut avoir influencé causalement b
C. que a et b sont sur la même machine

3. Deux événements aux horloges vectorielles [2,0] et [0,1] sont :

A. l'un avant l'autre

Quiz éclair

Quiz éclair (suite)

B. identiques

C. concurrents

4. 🔁 *Révision HEF-130, mission 01* : D'après Amdahl, si 90 % d'un programme est parallélisable, l'accélération maximale est :

A. environ ×10

B. infinie

C. ×90

5. 🔁 *Révision HEF-130, mission 09* : Une file d'attente sans limite sous surcharge conduit à :

A. une latence stable

B. une latence qui explose et une mémoire saturée

Quiz éclair

Quiz éclair (suite)

C. un gain de débit

Quiz éclair · réponse

Réponses

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

Je réfléchis

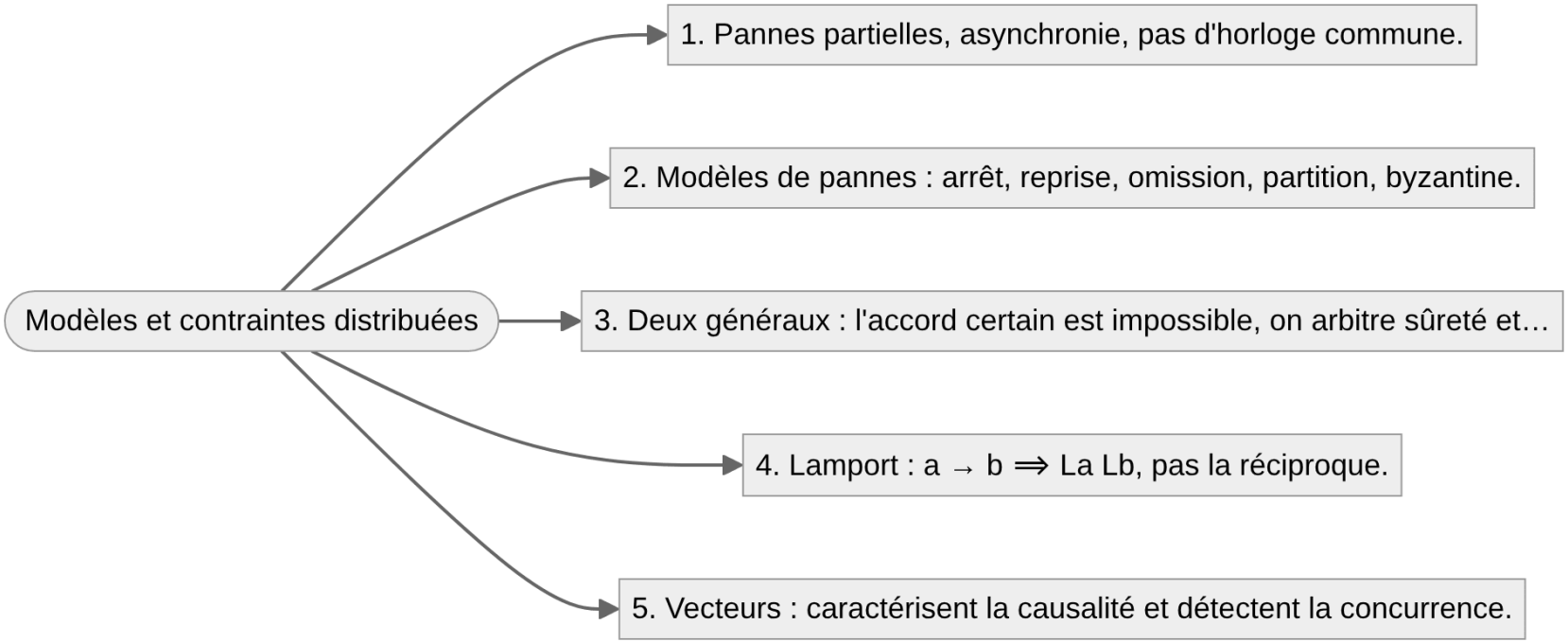
1. Quelle décision de ton système suppose aujourd'hui qu'une autre machine a « forcément » reçu ton message ?
2. Où ton système trie-t-il des événements par une heure physique issue de plusieurs machines ?

À retenir

1. Pannes partielles, asynchronie, pas d'horloge commune.
2. Modèles de pannes : arrêt, reprise, omission, partition, byzantine.
3. Deux généraux : l'accord certain est impossible, on arbitre sûreté et activité.
4. Lamport : $a \rightarrow b \Longrightarrow L(a) < L(b)$, pas la réciproque.
5. Vecteurs : caractérisent la causalité et détectent la concurrence.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Explorateur du distribué

Prochaine mission : RPC et communication

