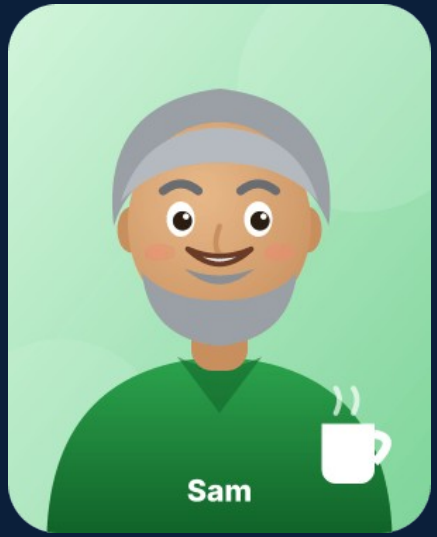
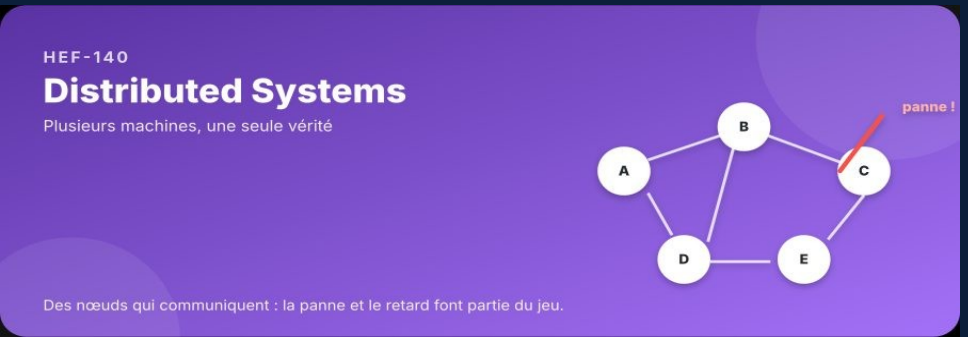
HEF-140 · MISSION 07 / 10

# Cohérence et consensus : panorama

*Quand plusieurs machines doivent s'accorder sur une seule vérité malgré les pannes, il faut un consensus. Et quand on y renonce, il faut savoir précisément ce qu'on perd.*




**Badge : Arbitre de la vérité**

115 min · avancé

# L'histoire

Deux clients de HashBox réservent la dernière place d'un cours en même temps, sur deux serveurs différents séparés par une coupure réseau. Chacun répond « confirmé ». Le lendemain, le cours a six inscrits pour cinq places. La direction demande : « Pourquoi n'avez-vous pas simplement refusé l'un des deux ? »

*« Parce que pendant la coupure, aucun serveur ne pouvait savoir ce que faisait l'autre, dit Sam. Pour refuser l'un des deux, il aurait fallu choisir entre répondre quand même (et risquer l'incohérence) ou refuser de répondre (et perdre la disponibilité). Ce choix a un nom : CAP. Et il existe des algorithmes — Paxos, Raft — qui permettent à un groupe de machines de décider de façon sûre, tant que la majorité vit. »*



# L'histoire (suite)

**Tu apprends le théorème CAP sans le caricaturer, les modèles de cohérence (linéarisable, séquentielle, causale, finale), le problème du consensus, et tu fais tourner une version simplifiée de Raft pour voir un leader se faire élire, des entrées se valider, et une partition ne pas casser la sûreté.**

Ta mission

# À la fin, je serai capable de…

1. énoncer précisément le théorème CAP et PACELC
2. distinguer linéarisabilité, cohérence séquentielle, causale, cohérence finale
3. définir le consensus (accord, validité, terminaison) et citer l'impossibilité FLP
4. comprendre élection de leader, réplication du journal et validation dans Raft
5. vérifier les propriétés de sûreté d'une exécution avec partitions et pannes

Défi d'échauffement

# Défi d'échauffement

Cinq amis doivent décider ensemble du restaurant du soir, uniquement par messages qui peuvent se perdre, et certains peuvent s'endormir. Règle : un restaurant n'est « choisi » que si la **majorité** (3 sur 5) l'a approuvé. Peut-on se retrouver avec deux restaurants « choisis » ? Que se passe-t-il si deux amis sont déconnectés ? Si trois sont déconnectés ?

Le concept

# Le théorème CAP, sans caricature

Trois propriétés d'un service de données répliqué :

| Lettre | Propriété | Signification précise |
|---|---|---|
| **C** | Cohérence (linéarisabilité) | toute lecture voit la dernière écriture, comme s'il n'y avait qu'une seule copie |
| **A** | Disponibilité | toute requête reçue par un nœud **non défaillant** obtient une réponse |
| **P** | Tolérance aux partitions | le système continue malgré des messages perdus entre nœuds |

Le concept

# Le théorème CAP, sans caricature (suite)

**CAP** : en présence d'une **partition**, on ne peut pas avoir à la fois C et A. Comme les partitions **arrivent** (on ne choisit pas), le vrai choix est : *pendant une partition, sacrifier la cohérence (AP) ou la disponibilité (CP)*.

- **CP** : les nœuds minoritaires **refusent** de répondre (ou répondent par une erreur) ; la vérité reste unique.
- **AP** : tous répondent, les copies **divergent**, on réconcilie après.

CAP n'est **pas** « choisis deux sur trois » à vie : sans partition, on peut avoir C et A ; et le choix se fait **par opération**, pas pour tout le système.

## PACELC : et quand tout va bien ?

**P**artition → choisir **A** ou **C** ; **E**lse (sinon) → choisir **L**atence ou **C**ohérence. Même sans panne, la cohérence forte coûte des allers-retours (latence). C'est le compromis de tous les jours.

Le concept

# Modèles de cohérence (du plus fort au plus faible)

| Modèle | Garantie | Exemple |
|---|---|---|
| **Linéarisable** | les opérations semblent atomiques et ordonnées selon le temps réel ; une lecture après une écriture terminée voit cette écriture | un compteur de stock, un verrou |
| **Séquentielle** | un ordre total unique, cohérent avec l'ordre de chaque client (pas nécessairement le temps réel) | journal global |
| **Causale** | si a → b, tous voient a avant b ; les concurrents peuvent être vus dans des ordres différents | commentaires et réponses |
| **« Lire ses écritures »** | un client voit ses propres écritures | profil utilisateur |
| **Finale** (*eventual*) | si plus d'écritures, les copies finissent par converger | DNS, compteurs de vues |

Le concept

# Modèles de cohérence (du plus fort au plus faible) (suite)

Plus le modèle est fort, plus il est simple à raisonner, et plus il coûte en latence et disponibilité.

## Le problème du consensus

Plusieurs processus proposent des valeurs ; ils doivent **décider** une valeur unique.

| Propriété | Sens |
|---|---|
| **Accord** (*agreement*) | deux processus corrects ne décident jamais de valeurs différentes |
| **Validité** | la valeur décidée a été proposée par quelqu'un |
| **Terminaison** | tout processus correct décide finalement |

Le concept

# Le problème du consensus (suite)

**Impossibilité FLP** (Fischer, Lynch, Paterson, 1985) : dans un système **asynchrone** où **un seul** processus peut tomber, aucun algorithme **déterministe** ne garantit à la fois accord et terminaison. En pratique, on garantit la **sûreté toujours** (accord, validité) et la **vivacité** (terminaison) **quand le réseau est suffisamment calme** (bornes de temps partielles, temporisations aléatoires).

## Raft : un consensus compréhensible

Raft décompose le problème :

1. **Élection du leader** : un seul leader par **terme** (numéro d'époque). Un suiveur qui n'entend plus de battements de cœur devient **candidat**, incrémente son terme, demande des votes. Un nœud vote **une seule fois par terme**. La **majorité** des voix fait un leader. Les délais d'élection sont **aléatoires** pour éviter les égalités.

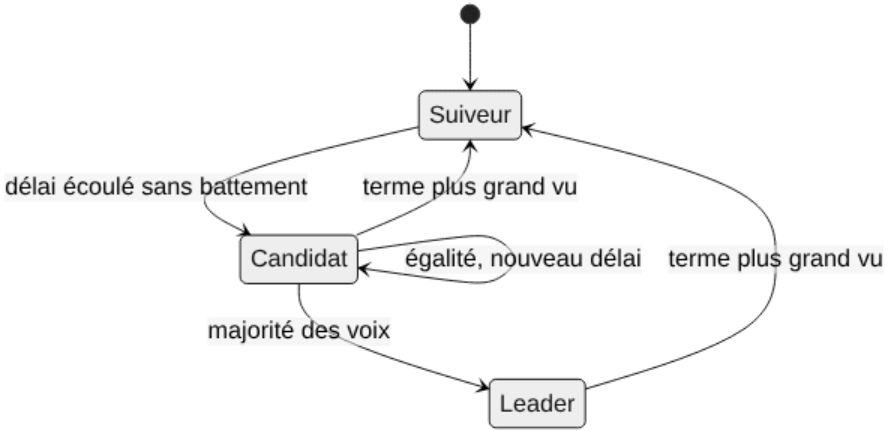
Le concept

# Raft : un consensus compréhensible (suite)

2. **Réplication du journal** : le leader reçoit les commandes, les ajoute à son journal, les envoie aux suiveurs (`AppendEntries`). Une entrée est **validée** (*committed*) quand elle est stockée sur une **majorité** ; le leader l'applique et informe les autres.
3. **Sûreté** :
   4. **un seul leader par terme** ;
   5. un candidat dont le journal est **moins à jour** ne gagne pas les voix : toute entrée validée est dans le journal de tout futur leader ;
   6. un suiveur dont le journal **diverge** est corrigé (les entrées conflictuelles non validées sont supprimées).

Le concept

# Raft : un consensus compréhensible (suite)




**Description du schéma** : un nœud commence suiveur ; sans nouvelles du leader il devient candidat ; avec la majorité des voix il devient leader ; s'il voit un terme plus grand il redevient suiveur.

Le concept

# Tailles de grappe et pannes tolérées

Une grappe de `N` nœuds nécessite une **majorité** `⌊N/2⌋ + 1`. Elle tolère `N − majorité` pannes :

| N | Majorité | Pannes tolérées |
|---|---|---|
| 3 | 2 | 1 |
| 4 | 3 | 1 (pas mieux que 3 !) |
| 5 | 3 | 2 |
| 7 | 4 | 3 |

On choisit un nombre **impair**. Une grappe ne tolère donc jamais la perte de la **majorité** : la minorité isolée **ne peut ni élire de leader ni valider** (choix CP).

Le concept

# Où trouve-t-on du consensus ?

Services de coordination (etcd, ZooKeeper, Consul) : configuration, élection de leader, verrous, découverte de services ; bases distribuées fortement cohérentes (journaux répliqués par Raft ou Paxos). On évite de l'employer pour le **volume** de données : on l'utilise pour les **métadonnées** et décisions critiques.

## Vérifier la cohérence : la linéarisabilité

Un historique d'opérations (avec heures de début et de fin) est **linéarisable** s'il existe un ordre séquentiel des opérations, **conforme au temps réel** (si a finit avant que b commence, a précède b), et valide pour la spécification (un registre rend la dernière valeur écrite). Les outils de test (Jepsen, Knossos) vérifient cela sur des exécutions réelles avec pannes injectées.

Le concept

# Vérifier la cohérence : la linéarisabilité (suite)

🧠 ***Astuce*** *: devant chaque donnée, demande : si deux copies divergent pendant une coupure, préfère-je répondre faux ou ne pas répondre ? Le stock d'un cours : ne pas répondre. Le nombre de vues : répondre.*

# Exemple pas à pas

**Problème** : faire tourner une version simplifiée de Raft (5 nœuds, réseau simulé avec délais et partitions) : élire un leader, valider des entrées, isoler le leader, voir la majorité élire un nouveau leader, constater qu'une entrée proposée à la minorité n'est jamais validée, réparer la partition, puis vérifier les propriétés de sûreté sur 60 scénarios aléatoires (partitions, pannes, redémarrages).

```
import random
from collections import defaultdict

class Noeud:
    def __init__(self, id_, ids, rng):
        self.id, self.ids, self.rng = id_, ids, rng
        self.terme, self.vote_pour, self.log, self.commit = 0, None, [], 0       #
    état DURABLE : terme, vote, journal
        self.redemarrer()
    def redemarrer(self):                                                        #
    état volatil (perdu à la panne)
        self.etat, self.votes, self.ecoule = "suiveur", set(), 0
```

# Exemple pas à pas (suite)

```
        self.delai = self.rng.randint(15, 30)
   # délai d'élection ALÉATOIRE
        self.next_index, self.match_index, self.depuis_hb = {}, {}, 0
    @property
    def majorite(self): return len(self.ids) // 2 + 1

class Cluster:
    def __init__(self, n=5, graine=1):
        self.rng = random.Random(graine)
        self.ids = list(range(n))
        self.noeuds = {i: Noeud(i, self.ids, self.rng) for i in self.ids}
        self.messages, self.t = [], 0                                  #
    (tick de livraison, message)
        self.arretes, self.groupes = set(), None                       #
    groupes : ensembles de nœuds qui peuvent communiquer
        self.leaders_par_terme = defaultdict(set)
    def peut_communiquer(self, a, b):
        if a in self.arretes or b in self.arretes: return False
        return self.groupes is None or any(a in g and b in g for g in self.groupes
```

# Exemple pas à pas (suite)

```
)
 def envoyer(self, de, a, **msg):
     if self.peut_communiquer(de, a):
         self.messages.append((self.t + self.rng.randint(1, 2), dict(msg, de=de
, a=a)))     # délai de 1 à 2 ticks
 def devenir_suiveur(self, n, terme):
     n.terme, n.etat, n.vote_pour, n.votes = terme, "suiveur", None, set()
 def demarrer_election(self, n):
     n.etat, n.terme, n.vote_pour, n.votes, n.ecoule = "candidat", n.terme + 1,
 n.id, {n.id}, 0
     n.delai = self.rng.randint(15, 30)
     li, lt = len(n.log), (n.log[-1][0] if n.log else 0)
     for p in self.ids:
         if p != n.id: self.envoyer(n.id, p, type="RV", terme=n.terme, idx=li,
tl=lt)
 def devenir_leader(self, n):
     n.etat = "leader"; self.leaders_par_terme[n.terme].add(n.id)
     n.log.append((n.terme, None))
# entrée vide : permet de valider le terme courant
```

# Exemple pas à pas (suite)

```
        n.next_index = {p: len(n.log) for p in self.ids}; n.match_index = {p: 0 fo
    r p in self.ids}
        n.depuis_hb = 99
     def envoyer_entrees(self, n):
        for p in self.ids:
            if p == n.id: continue
            prev = n.next_index[p] - 1
            self.envoyer(n.id, p, type="AE", terme=n.terme, prev=prev, prev_t=(n.l
    og[prev - 1][0] if prev > 0 else 0),
                         entrees=list(n.log[prev:]), commit=n.commit)
     def traiter(self, n, m):
        if m["terme"] > n.terme and m["type"] in ("RV", "AE"): self.devenir_suiveu
    r(n, m["terme"])
        if m["type"] == "RV":
            dernier = n.log[-1][0] if n.log else 0
            journal_ok = m["tl"] > dernier or (m["tl"] == dernier and m["idx"] >=
    len(n.log))        # le candidat est-il au moins aussi à jour ?
            accorde = m["terme"] == n.terme and n.vote_pour in (None, m["de"]) and
     journal_ok          # UN vote par terme
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
            if accorde: n.vote_pour, n.ecoule = m["de"], 0
            self.envoyer(n.id, m["de"], type="RVR", terme=n.terme, ok=accorde)
        elif m["type"] == "RVR":
            if m["terme"] > n.terme: self.devenir_suiveur(n, m["terme"])
            elif n.etat == "candidat" and m["terme"] == n.terme and m["ok"]:
                n.votes.add(m["de"])
                if len(n.votes) >= n.majorite: self.devenir_leader(n)
        elif m["type"] == "AE":
            if m["terme"] < n.terme: self.envoyer(n.id, m["de"], type="AER", terme
    =n.terme, ok=False, match=0); return
            n.etat, n.ecoule = "suiveur", 0
            if m["prev"] > 0 and (len(n.log) < m["prev"] or n.log[m["prev"] - 1][0
    ] != m["prev_t"]):
                self.envoyer(n.id, m["de"], type="AER", terme=n.terme, ok=False, m
    atch=0); return          # journaux divergents : le leader reculera
            idx = m["prev"]
            for e in m["entrees"]:
                idx += 1
                if len(n.log) >= idx and n.log[idx - 1][0] != e[0]: del n.log[idx
```

# Exemple pas à pas (suite)

```
- 1:]                    # supprime les entrées conflictuelles
            if len(n.log) < idx: n.log.append(e)
        n.commit = max(n.commit, min(m["commit"], m["prev"] + len(m["entrees"]
)))
        self.envoyer(n.id, m["de"], type="AER", terme=n.terme, ok=True, match=
m["prev"] + len(m["entrees"]))
    elif m["type"] == "AER":
        if m["terme"] > n.terme: self.devenir_suiveur(n, m["terme"]); return
        if n.etat != "leader" or m["terme"] != n.terme: return
        p = m["de"]
        if m["ok"]:
            n.match_index[p] = max(n.match_index[p], m["match"]); n.next_index
[p] = n.match_index[p] + 1
            for N in range(len(n.log), n.commit, -1):
# valide la plus grande entrée stockée sur une MAJORITÉ
                if n.log[N - 1][0] == n.terme and 1 + sum(1 for q in self.ids
if q != n.id and n.match_index[q] >= N) >= n.majorite:
                    n.commit = N; break
        else:
```

# Exemple pas à pas (suite)

```
                n.next_index[p] = max(1, n.next_index[p] - 1)
    def tick(self):
        self.t += 1
        dus = [m for t, m in self.messages if t <= self.t]; self.messages = [(t, m
    ) for t, m in self.messages if t > self.t]
        for m in dus:
            if m["a"] not in self.arretes: self.traiter(self.noeuds[m["a"]], m)
        for n in self.noeuds.values():
            if n.id in self.arretes: continue
            if n.etat == "leader":
                n.depuis_hb += 1
                if n.depuis_hb >= 5: n.depuis_hb = 0; self.envoyer_entrees(n)
            else:
                n.ecoule += 1
                if n.ecoule >= n.delai: self.demarrer_election(n)
    def run(self, ticks):
        for _ in range(ticks): self.tick()
    def leader_actuel(self):
        c = [n for n in self.noeuds.values() if n.etat == "leader" and n.id not in
```

# Exemple pas à pas (suite)

```
    self.arretes]
        return max(c, key=lambda n: n.terme) if c else None
    def proposer(self, commande):
        l = self.leader_actuel()
        if l is None: return None
        l.log.append((l.terme, commande)); return l.id, len(l.log)

# ---------------- scénario pas à pas ----------------
c = Cluster(5, graine=1)
c.run(100)
l = c.leader_actuel()
print(f"1. élection : le nœud {l.id} est leader au terme {l.terme}")
assert l is not None and all(len(ids) <= 1 for ids in c.leaders_par_terme.values()
   )
for cmd in ("inscription:ada", "inscription:bob", "inscription:cleo"): c.proposer(
   cmd)
c.run(40)
print("2. après 3 commandes, (nœud, taille du journal, entrées validées) :", [(n.i
   d, len(n.log), n.commit) for n in c.noeuds.values()])
```

# Exemple pas à pas (suite)

```
assert all(len(n.log) == 4 and n.commit == 4 for n in c.noeuds.values())
   # 1 entrée vide + 3 commandes, partout

ancien = l.id
c.groupes = [{ancien}, set(c.ids) - {ancien}]
     # PARTITION : le leader est isolé de la majorité
c.run(100)
nouveau = max((n for n in c.noeuds.values() if n.etat == "leader" and n.id != anci
   en), key=lambda n: n.terme)
print(f"3. partition : la majorité élit le nœud {nouveau.id} (terme {nouveau.terme
   }) ; l'ancien leader {ancien} se croit toujours leader (terme {c.noeuds[ancien]
   .terme})")
assert nouveau.terme > c.noeuds[ancien].terme and c.noeuds[ancien].etat == "leader
   "
c.noeuds[ancien].log.append((c.noeuds[ancien].terme, "inscription:dan (MINORITÉ)")
   )      # un client écrit sur l'ancien leader
nouveau.log.append((nouveau.terme, "inscription:eve (MAJORITÉ)"))
c.run(40)
print(f"4. entrées validées : minorité (ancien leader) = {c.noeuds[ancien].commit}
```

# Exemple pas à pas (suite)

```
   , majorité (nouveau leader) = {nouveau.commit}")
assert c.noeuds[ancien].commit == 4 and nouveau.commit == 6
           # l'écriture en minorité n'est JAMAIS validée

c.groupes = None; c.run(100)
           # la partition est réparée
journaux = {tuple(e[1] for e in n.log) for n in c.noeuds.values()}
print("5. après réparation :", journaux)
assert len(journaux) == 1 and "inscription:dan (MINORITÉ)" not in next(iter(journa
   ux))        # l'entrée de la minorité a été écrasée
assert c.noeuds[ancien].etat == "suiveur"

# ---------------- propriétés de sûreté sur 60 scénarios aléatoires --------------
   --
def scenario(graine, ticks=600):
    c = Cluster(5, graine); rng = random.Random(graine * 7 + 1)
    valide, nb = {}, 0
    for t in range(ticks):
        if t % 60 == 0:
```

# Exemple pas à pas (suite)

```
            r = rng.random()
            if r < 0.3: k = rng.randint(1, 2); a = set(rng.sample(c.ids, k)); c.gr
oupes = [a, set(c.ids) - a]
            elif r < 0.5: c.groupes = None
            elif r < 0.7 and len(c.arretes) < 2: c.arretes.add(rng.choice(c.ids))
            elif r < 0.9 and c.arretes:
                x = rng.choice(sorted(c.arretes)); c.arretes.discard(x); c.noeuds[
x].redemarrer()
        if t % 17 == 0 and c.proposer(f"cmd{nb}"): nb += 1
        c.tick()
        for n in c.noeuds.values():
            for i in range(1, n.commit + 1):
                e = n.log[i - 1]
                assert valide.setdefault(i, e) == e, (graine, t, "une entrée VALID
ÉE a changé")       # sûreté : le passé validé ne change jamais
     assert all(len(ids) <= 1 for ids in c.leaders_par_terme.values()), (graine, "d
eux leaders dans un même terme")
     return len(valide)
```

# Exemple pas à pas (suite)

```
resultats = [scenario(g) for g in range(60)]
print(f"6. 60 scénarios avec partitions et pannes : sûreté respectée ; entrées val
   idées par scénario : min {min(resultats)}, moyenne {sum(resultats) / 60:.0f}, m
   ax {max(resultats)}")
assert min(resultats) > 0
```

**Lecture** :

1. Un leader est **élu** par la majorité (au plus un par terme), puis les commandes sont répliquées et **validées** sur les 5 nœuds.
2. Pendant la **partition**, la majorité (4 nœuds) élit un **nouveau leader à un terme supérieur**. L'ancien leader, isolé, se croit toujours leader mais **ne peut rien valider** : l'écriture proposée à la minorité reste à 4 entrées validées. C'est le choix **CP** : la minorité sacrifie la disponibilité pour préserver la vérité.

# Exemple pas à pas (suite)

3. À la **réparation**, l'ancien leader voit un terme supérieur, **redevient suiveur**, et son entrée non validée est **écrasée** : tous les journaux deviennent identiques.
4. Sur 60 scénarios aléatoires (partitions, arrêts, redémarrages), **aucune entrée validée ne change** et il n'y a **jamais deux leaders dans le même terme** : les propriétés de sûreté tiennent, même quand la **vivacité** est parfois perdue (pas de leader pendant un moment).

# À toi, avec aide

Pour des grappes Raft de 3, 4, 5, 6, 7 et 9 nœuds :

1. calcule la majorité et le nombre de pannes tolérées ;
2. explique pourquoi 4 nœuds ne valent pas mieux que 3 (et 6 pas mieux que 5) ;
3. pour une grappe de 5 nœuds répartis en 3 centres (2 + 2 + 1), quelles pannes de centre sont tolérées ? Et si les centres sont répartis 3 + 1 + 1 ?

À toi, avec aide · réponse

# Indices progressifs

## Indice 1

Majorité = N // 2 + 1 ; pannes tolérées = N − majorité.

## Indice 2

Pour les centres : la grappe tolère la perte d'un centre si les nœuds restants forment une majorité.

## Indice 3

3 + 1 + 1 : perdre le gros centre fait perdre 3 nœuds sur 5.

# Réponse

```
for n in (3, 4, 5, 6, 7, 9):
    maj = n // 2 + 1
    print(f"N = {n} : majorité {maj}, pannes tolérées {n - maj}")
def tolere_centre(repartition):
    n = sum(repartition); maj = n // 2 + 1
    return [i for i, taille in enumerate(repartition) if n - taille >= maj]
print("2 + 2 + 1 :", tolere_centre([2, 2, 1]), "| 3 + 1 + 1 :", tolere_centre([3,
   1, 1]))
assert [n - (n // 2 + 1) for n in (3, 4, 5, 6, 7, 9)] == [1, 1, 2, 2, 3, 4]
assert tolere_centre([2, 2, 1]) == [0, 1, 2] and tolere_centre([3, 1, 1]) == [1, 2
   ]
```

2. 4 nœuds exigent une majorité de 3 et tolèrent **1** panne, exactement comme 3 nœuds — avec un nœud en plus à entretenir et un risque d'égalité 2 contre 2 lors d'une partition. On choisit un **nombre impair**.

# Réponse (suite)

3. **2 + 2 + 1** : la perte de n'importe quel centre laisse 3 ou 4 nœuds : **tout centre peut tomber**. **3 + 1 + 1** : la perte du gros centre laisse 2 nœuds sur 5 : **plus de majorité**. La répartition **équilibrée** entre centres compte autant que le nombre de nœuds.

# À toi, en solo

**Vérificateur de linéarisabilité.** Écris `est_lineaire(historique)` pour un **registre** (lecture/écriture d'une valeur entière, valeur initiale 0). Chaque opération a un client, un type (`ecrire(v)` ou `lire() → v`), une heure de début et une heure de fin.

1. recherche en profondeur : à chaque étape, choisis une opération dont **aucune autre opération restante n'a fini avant son début** (elle peut être ordonnée maintenant), applique-la (lecture : la valeur lue doit égaler l'état courant) ;
2. teste cinq historiques :
   3. (a) écriture(1) [0,2] ; lecture→1 [3,4] : linéarisable ;
   4. (b) écriture(1) [0,2] ; lecture→0 [3,4] : **non** linéarisable (lecture périmée après une écriture terminée) ;
   5. (c) écriture(1) [0,5] ; lecture→0 [1,2] ; lecture→1 [3,4] : linéarisable (lecture concurrente de l'écriture) ;

# À toi, en solo (suite)

   6. (d) écriture(1) [0,5] ; lecture→1 [1,2] ; lecture→0 [3,4] : **non** linéarisable (la valeur « revient en arrière ») ;
   7. (e) deux écritures concurrentes (1 et 2) et deux lectures successives qui voient 2 puis 1 : à toi de décider, avec le vérificateur ;
8. génère 300 historiques aléatoires d'un registre **réellement atomique** (exécution séquentielle simulée avec durées aléatoires) et vérifie qu'ils sont tous linéarisables ; puis injecte des **lectures périmées** et mesure la part détectée.

À toi, en solo · réponse

# Pistes de correction

```
import random
from functools import lru_cache

def est_lineaire(operations):
    ops = tuple(operations)                                          # (cl
   ient, type, valeur, debut, fin)
    @lru_cache(maxsize=None)
    def chercher(restantes, etat):
        if not restantes: return True
        for i in restantes:
            _, type_, valeur, debut, _ = ops[i]
            # i peut passer maintenant si aucune autre opération restante n'a FINI
    avant son début
            if any(ops[j][4] < debut for j in restantes if j != i): continue
            if type_ == "w": nouvel_etat = valeur
            else:
                if valeur != etat: continue
                nouvel_etat = etat
            if chercher(restantes - {i}, nouvel_etat): return True
```

# Pistes de correction (suite)

```
        return False
    return chercher(frozenset(range(len(ops))), 0)

W = lambda c, v, d, f: (c, "w", v, d, f)
R = lambda c, v, d, f: (c, "r", v, d, f)
a = [W("A", 1, 0, 2), R("B", 1, 3, 4)]
b = [W("A", 1, 0, 2), R("B", 0, 3, 4)]
c_ = [W("A", 1, 0, 5), R("B", 0, 1, 2), R("B", 1, 3, 4)]
d = [W("A", 1, 0, 5), R("B", 1, 1, 2), R("B", 0, 3, 4)]
e = [W("A", 1, 0, 5), W("C", 2, 0, 5), R("B", 2, 6, 7), R("B", 1, 8, 9)]
assert [est_lineaire(x) for x in (a, b, c_, d, e)] == [True, False, True, False, F
   alse]
print("(e) non linéarisable : après avoir lu 2, la valeur ne peut plus redevenir 1
    (les deux écritures sont terminées)")

def historique_atomique(rng, n_clients=3, n_ops=9):
    # registre ATOMIQUE : chaque opération prend effet à un instant précis ; sa du
   rée englobe cet instant
    etat, t, ops = 0, 0.0, []
```

À toi, en solo · réponse

# Pistes de correction (suite)

```
    for i in range(n_ops):
        t += rng.uniform(0.5, 2); instant = t
        client = f"C{rng.randrange(n_clients)}"
        debut, fin = instant - rng.uniform(0, 1.5), instant + rng.uniform(0, 1.5)
        if rng.random() < 0.5:
            v = rng.randint(1, 5); etat = v; ops.append((client, "w", v, debut, fi
   n))
        else:
            ops.append((client, "r", etat, debut, fin))
    return ops

rng = random.Random(1)
assert all(est_lineaire(historique_atomique(rng)) for _ in range(300))
print("300 historiques d'un registre atomique : tous linéarisables")
detectees = total = 0
for _ in range(300):
    h = historique_atomique(rng)
    lectures = [i for i, o in enumerate(h) if o[1] == "r"]
    if not lectures: continue
```

À toi, en solo · réponse

# Pistes de correction (suite)

```
    i = rng.choice(lectures); o = h[i]
    h[i] = (o[0], "r", (o[2] + rng.randint(1, 4)) % 6, o[3], o[4])
   # lecture « périmée » ou fausse
    total += 1; detectees += (not est_lineaire(h))
print(f"lectures corrompues détectées : {detectees}/{total}")
assert detectees / total > 0.5
```

Le vérificateur **explore des ordres** compatibles avec le temps réel. C'est exactement ce que font les outils de test de bases de données distribuées (Jepsen/Knossos) : ils injectent pannes et partitions, enregistrent l'historique observé, puis vérifient qu'**au moins un ordre linéarisable** l'explique. Les lectures corrompues ne sont pas toutes détectées : certaines valeurs fausses restent compatibles avec un ordre possible.

# Pièges classiques

| Piège | Pourquoi c'est un problème | Comment l'éviter |
|---|---|---|
| **« CAP : choisis deux sur trois »** | Simplification fausse : P n'est pas optionnel | Pendant une partition : C ou A, opération par opération |
| **Confondre disponibilité et cohérence** | Réponses rapides mais fausses, ou refus injustifiés | Décider par donnée : stock (C), compteur de vues (A) |
| **Grappe de consensus en nombre pair, ou répartie sur un seul site** | Pas de gain, ou perte de majorité sur panne d'un site | Nombre impair, centres équilibrés |
| **Utiliser le consensus pour de gros volumes** | Latence et débit insuffisants | Consensus pour les métadonnées et décisions critiques |

Mini-boss

# Mini-boss

**Banc d'essai de consensus.** Étends le mini-Raft de la mission :

1. ajoute des **clients** qui envoient des commandes avec identifiant (idempotence, mission 4) et reçoivent une confirmation **uniquement** quand l'entrée est validée ;
2. implémente une machine à états « compteur de places par cours » appliquée à partir des entrées validées, avec capacité maximale ;
3. injecte pannes, partitions, pertes et doublons de messages (réseau configurable) ;
4. vérifie les invariants sur 200 graines : un seul leader par terme ; les entrées validées ne changent jamais ; **aucun cours n'est surbooké** ; aucune commande confirmée n'est perdue ; chaque commande est appliquée **au plus une fois** malgré les relances ;
5. mesure la **vivacité** : délai d'élection moyen, proportion de commandes confirmées, durée d'indisponibilité pendant une partition ;
6. compare avec une variante **sans consensus** (deux nœuds acceptent chacun les inscriptions pendant la partition) et montre le surbooking ;

# Mini-boss (suite)

7. rédige un tableau CAP : pour chaque scénario, ce que le système a choisi (C ou A) et ce que cela a coûté.

Mini-boss · réponse

# Pistes de correction

Un bon travail : une file de commandes avec identifiants, des clients qui relancent tant qu'ils n'ont pas de confirmation, une machine à états appliquée sur `commit` avec déduplication par identifiant; invariants vérifiés dans un moniteur; mesure de la disponibilité (la minorité ne confirme rien, la majorité oui); variante AP montrant le surbooking en cas de partition; tableau CAP clair. Le message clé : Raft garantit la sûreté même quand la vivacité est suspendue; le prix de la cohérence forte est l'indisponibilité de la minorité.

Quiz éclair

# Quiz éclair

**1.** Pendant une partition réseau, un système CP :

A. refuse de répondre côté minoritaire pour préserver la cohérence
B. répond partout avec des copies divergentes
C. ignore la partition

**2.** Dans Raft, une entrée est validée (committed) quand elle est stockée sur :

A. tous les nœuds
B. une majorité de nœuds
C. un seul nœud

**3.** Une grappe de 5 nœuds de consensus tolère :

A. 3 pannes

Quiz éclair

# Quiz éclair (suite)

B. 5 pannes
C. 2 pannes

**4.** 🔁 *Révision mission 06* : Avec `hash(clé) mod N`, ajouter un serveur :

A. déplace la majorité des clés
B. ne déplace aucune clé
C. déplace exactement 1/N des clés

**5.** 🔁 *Révision mission 06* : Le hachage cohérent avec nœuds virtuels :

A. supprime le besoin de répliquer
B. limite les déplacements à environ 1/(N+1) des clés et équilibre la charge
C. empêche toute panne

Quiz éclair · réponse

# Réponses

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

# Je réfléchis

1. Pour quelle donnée de ton système préférerais-tu une erreur à une réponse potentiellement fausse ?
2. Combien de tes services dépendent d'une seule instance « qui ne tombe jamais » ?

À retenir

# À retenir

1. CAP : pendant une partition, choisir C ou A ; PACELC ajoute latence contre cohérence.
2. Modèles : linéarisable, séquentiel, causal, finale.
3. Consensus : accord, validité, terminaison ; FLP en asynchrone pur.
4. Raft : élection (un leader par terme), journal répliqué, validation à la majorité.
5. Majorité nécessaire : nombre impair de nœuds, centres équilibrés.

MISSION ACCOMPLIE

# +100 XP

## Badge débloqué : Arbitre de la vérité

Prochaine mission : Files et événements

