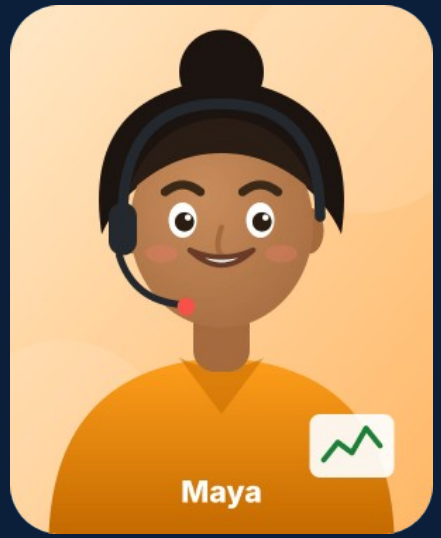
HEF-130 · MISSION 06 / 10

# Réseau et latence

*La vitesse de la lumière est une limite que personne n'optimisera. La seule solution : faire moins d'allers-retours.*





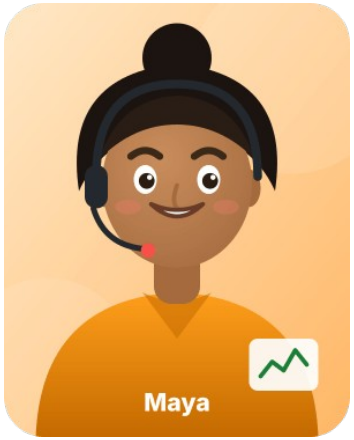


**Badge : Dompteur de distances**

100 min · avancé

# L'histoire

La page « Mes cours » de HashBox s'affiche en 400 ms depuis le bureau de Kodi. Pour une formatrice à Montréal, 4,2 secondes. Le serveur est pourtant le même, et il répond en 20 ms. Sam ouvre l'outil de réseau du navigateur : 47 requêtes successives, chacune attendant la précédente, chacune coûtant un aller-retour de 90 ms.




*« 47 allers-retours × 90 ms = 4,2 secondes, dit Sam. Ni le serveur ni la bande passante n'y sont pour quelque chose. C'est la distance, multipliée par le nombre de fois qu'on la parcourt. Le remède est de parcourir moins souvent, pas plus vite. »*

**Tu apprends à modéliser un temps de chargement (DNS, TCP, TLS, requêtes), à réutiliser les connexions, à regrouper les requêtes, à compter les connexions réelles sur un serveur, et à éviter de retransférer ce qui n'a pas changé.**

Ta mission

# À la fin, je serai capable de…

1. décomposer le coût d'une requête en allers-retours (DNS, TCP, TLS, requête/réponse)
2. distinguer latence, bande passante et produit bande passante × délai
3. réutiliser les connexions (keep-alive) et mesurer leur effet en comptant les connexions
4. regrouper les requêtes (lots) et paralléliser pour réduire le nombre d'allers-retours séquentiels
5. utiliser la validation de cache HTTP (ETag, 304) et la compression pour réduire les octets

Défi d'échauffement

# Défi d'échauffement

Tu commandes 50 articles dans un magasin à l'autre bout du pays. Soit tu passes 50 appels, un par article, en attendant chaque livraison avant d'appeler pour le suivant ; soit tu passes une seule commande de 50 articles. Chaque appel et chaque livraison prennent 2 jours. Combien de jours dans chaque cas ? Qu'est-ce qui compte le plus : la taille de chaque colis ou le nombre d'appels ?

Le concept

# Latence et bande passante

| Notion | Sens | Analogie |
|---|---|---|
| **Latence / délai aller-retour (RTT)** | temps pour qu'un message aille et revienne | durée d'un trajet |
| **Bande passante (débit)** | quantité de données par seconde | largeur de l'autoroute |
| **Produit bande passante × RTT** | données « en vol » pour remplir le tuyau | nombre de voitures sur la route |

Pour de **petites** requêtes, la latence domine (peu importe la largeur de la route, c'est la distance qui compte). Pour de **gros** transferts, la bande passante domine.

Le concept

# Combien coûte une requête ?

Pour une première requête vers un site HTTPS, avant le moindre octet utile :

| Étape | Allers-retours |
|---|---|
| Résolution DNS | 1 RTT (si non en cache) |
| Connexion TCP (poignée de main à trois temps) | 1 RTT |
| Négociation TLS 1.3 | 1 RTT (TLS 1.2 : 2) |
| La requête HTTP et sa réponse | 1 RTT |
| **Total** | **≈ 4 RTT** |

Avec 90 ms de RTT : **360 ms** avant d'avoir la première réponse, même si le serveur répond instantanément.

Le concept

# Réutiliser les connexions

- **Keep-alive** (HTTP/1.1) : la connexion reste ouverte pour plusieurs requêtes → plus de poignée de main répétée.
- **Pool de connexions** : un client garde N connexions prêtes.
- **HTTP/2 et HTTP/3** : plusieurs requêtes **en même temps** sur une seule connexion (multiplexage), compression des en-têtes.
- **Reprise de session TLS / 0-RTT** : raccourcit la négociation pour une reconnexion.

## Réduire le nombre d'allers-retours séquentiels

| Technique | Effet |
|---|---|
| **Regrouper** (*batching*) : une requête qui demande 50 éléments | 50 RTT → 1 RTT |

Le concept

# Réduire le nombre d'allers-retours séquentiels (suite)

| Technique | Effet |
|---|---|
| **Paralléliser** : lancer les requêtes indépendantes en même temps | RTT × N → RTT × (N / parallélisme) |
| **Pré-charger / précharger** (*prefetch*) | masque la latence |
| **Rapprocher** : CDN, régions proches, cache en périphérie | réduit le RTT lui-même |
| **Éviter les cascades** : A attend B qui attend C | rendre les appels indépendants |

Le **N+1 réseau** : une requête qui récupère une liste, puis une requête **par élément** pour ses détails. À proscrire (cf. N+1 en base de données, mission 7).

## Réduire les octets

| Technique | Effet |
|---|---|
| **Compression** (gzip, brotli) | texte/JSON : −70 à −90 % |

Le concept

# Réduire les octets (suite)

| Technique | Effet |
|---|---|
| **Cache HTTP** (`Cache-Control: max-age`) | pas de requête du tout |
| **Validation** (`ETag` + `If-None-Match`) → **304 Not Modified** | un aller-retour mais **aucun corps** |
| **Formats compacts** | JSON → binaire |
| **Champs à la demande** | ne renvoyer que ce qui est demandé |
| **Images adaptées** | format et taille appropriés |

## Le produit bande passante × délai et la fenêtre TCP

Pour remplir un tuyau de 100 Mbit/s avec un RTT de 100 ms, il faut 1,25 Mo « en vol ». Si la fenêtre TCP est de 64 Ko, le débit est plafonné à `fenêtre / RTT` = **5,2 Mbit/s**, quelle que soit la bande passante. D'où l'intérêt des réglages de fenêtre et de protocoles modernes pour les liaisons longues.

Le concept

# Le démarrage lent de TCP

TCP augmente progressivement son débit (*slow start*) : les premières centaines de Ko d'une connexion neuve sont envoyées plus lentement. Réutiliser une connexion déjà « chaude » est donc doublement avantageux.

## Mesurer

Outils : outils de développement du navigateur (cascade réseau), `curl -w` (temps DNS, connexion, TLS, premier octet), `ping`, `traceroute`. Chercher : combien de requêtes ? lesquelles sont séquentielles ? combien d'octets ?

🧠 ***Astuce** : devant une page lente, compte les **allers-retours séquentiels** sur le chemin critique. Réduire ce nombre vaut presque toujours plus qu'accélérer le serveur.*

# Exemple pas à pas

**Problème** : (1) modéliser le temps de chargement selon la stratégie, (2) compter réellement les connexions d'un serveur local avec et sans keep-alive, (3) montrer le gain d'une validation de cache `304` et d'une compression.

```
import gzip, http.client, json, threading
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

# ---------------- 1. modèle de temps de chargement ----------------
def temps_chargement(n_ressources, rtt, taille_ko, bande_mbps, connexions_parallel
   es=1, nouvelle_connexion=True, multiplex=False):
    poignee = 3 * rtt if nouvelle_connexion else 0                          #
   DNS + TCP + TLS 1.3
    transfert_total = n_ressources * taille_ko * 8 / 1000 / bande_mbps
   # la bande passante est partagée : le total ne change pas
    if multiplex:
        return poignee + rtt + transfert_total
   # toutes les requêtes en même temps sur une connexion
    vagues = -(-n_ressources // connexions_paralleles)
```

# Exemple pas à pas (suite)

```
    # allers-retours successifs
     return poignee + vagues * rtt + transfert_total

RTT, TAILLE, BANDE = 0.090, 20, 50                                      #
     90 ms, 20 Ko par ressource, 50 Mbit/s
strategies = {
    "47 requêtes en série": temps_chargement(47, RTT, TAILLE, BANDE),
    "6 connexions parallèles": temps_chargement(47, RTT, TAILLE, BANDE, connexions
    _paralleles=6),
    "HTTP/2 multiplexé": temps_chargement(47, RTT, TAILLE, BANDE, multiplex=True),
    "1 requête groupée (47 × 20 Ko)": temps_chargement(1, RTT, TAILLE * 47, BANDE)
    ,
}
for nom, t in strategies.items():
    print(f"  {nom:32s} {t * 1000:7.0f} ms")
assert strategies["47 requêtes en série"] > 3.5 * strategies["6 connexions parallè
    les"]
assert strategies["1 requête groupée (47 × 20 Ko)"] < strategies["HTTP/2 multiplex
    é"] + 0.05
```

# Exemple pas à pas (suite)

```
assert strategies["47 requêtes en série"] > 8 * strategies["1 requête groupée (47
   × 20 Ko)"]

# ---------------- 2. serveur local qui compte les connexions ----------------
class Serveur(ThreadingHTTPServer):
    daemon_threads = True
    def __init__(self, *a, **k):
        super().__init__(*a, **k); self.connexions = 0; self.octets = 0
    def get_request(self):
        c = super().get_request(); self.connexions += 1; return c

DONNEES = json.dumps([{"membre": f"m{i}", "cours": f"HEF-{i % 90:03d}", "statut":
   "inscrit"} for i in range(800)]).encode()
ETAG = '"v1"'

class Handler(BaseHTTPRequestHandler):
    protocol_version = "HTTP/1.1"                                          #
    keep-alive possible
    def do_GET(self):
```

Exemple pas à pas

# Exemple pas à pas (suite)

```
        if self.headers.get("If-None-Match") == ETAG:
            self.send_response(304); self.send_header("ETag", ETAG); self.send_hea
   der("Content-Length", "0"); self.end_headers(); return
        corps = DONNEES
        if "gzip" in self.headers.get("Accept-Encoding", ""):
            corps = gzip.compress(DONNEES, 6)
        self.send_response(200)
        self.send_header("ETag", ETAG); self.send_header("Content-Type", "applicat
   ion/json"); self.send_header("Content-Length", str(len(corps)))
        if "gzip" in self.headers.get("Accept-Encoding", ""): self.send_header("Co
   ntent-Encoding", "gzip")
        self.end_headers(); self.wfile.write(corps)
        self.server.octets += len(corps)
    def log_message(self, *a): pass

serveur = Serveur(("127.0.0.1", 0), Handler)
threading.Thread(target=serveur.serve_forever, daemon=True).start()
port = serveur.server_address[1]
```

# Exemple pas à pas (suite)

```
# une nouvelle connexion à chaque requête
for _ in range(20):
    c = http.client.HTTPConnection("127.0.0.1", port); c.request("GET", "/"); c.ge
   tresponse().read(); c.close()
nouvelles = serveur.connexions
# une connexion réutilisée
avant = serveur.connexions
c = http.client.HTTPConnection("127.0.0.1", port)
for _ in range(20):
    c.request("GET", "/"); c.getresponse().read()
c.close()
reutilisee = serveur.connexions - avant
print(f"20 requêtes → connexions ouvertes : sans réutilisation {nouvelles}, avec k
   eep-alive {reutilisee}")
assert nouvelles == 20 and reutilisee == 1

# ---------------- 3. compression et validation de cache ----------------
def octets_pour(entetes):
    serveur.octets = 0
```

# Exemple pas à pas (suite)

```
    c = http.client.HTTPConnection("127.0.0.1", port); c.request("GET", "/", heade
   rs=entetes)
    r = c.getresponse(); r.read(); c.close()
    return r.status, serveur.octets

s_brut, o_brut = octets_pour({})
s_gzip, o_gzip = octets_pour({"Accept-Encoding": "gzip"})
s_304, o_304 = octets_pour({"If-None-Match": ETAG, "Accept-Encoding": "gzip"})
print(f"sans compression : {o_brut} octets | gzip : {o_gzip} octets | revalidation
    (304) : {o_304} octets (statut {s_304})")
assert s_brut == s_gzip == 200 and s_304 == 304
assert o_gzip < o_brut / 5 and o_304 == 0
serveur.shutdown()

# ---------------- 4. fenêtre TCP et produit bande passante × délai --------------
   --
def debit_max_mbps(fenetre_ko, rtt_s):
    return fenetre_ko * 1024 * 8 / rtt_s / 1e6
print(f"fenêtre 64 Ko, RTT 100 ms → plafond de {debit_max_mbps(64, 0.1):.1f} Mbit/
```

# Exemple pas à pas (suite)

```
    s, même sur un lien de 1 Gbit/s")
assert round(debit_max_mbps(64, 0.1), 1) == 5.2
```

**Lecture** :

1. À RTT égal, **47 requêtes en série** coûtent environ **4,6 secondes** ; 6 connexions parallèles divisent par plus de 4 (≈ 1,1 s) ; le multiplexage et **une requête groupée** descendent à ≈ 0,5 s. Même bande passante, même serveur : seul le nombre d'allers-retours séquentiels a changé.
2. Le serveur **compte les connexions** : 20 requêtes avec une connexion neuve chaque fois = 20 connexions ; avec réutilisation (keep-alive) = **1 seule**.
3. La **compression gzip** divise le volume par plus de cinq ; la **revalidation** (`If-None-Match` → `304`) réduit la réponse à **zéro octet de corps** (un aller-retour reste nécessaire, mais très léger).
4. La **fenêtre TCP** de 64 Ko plafonne le débit à ≈ 5 Mbit/s pour 100 ms de RTT : la bande passante achetée ne sert à rien si la fenêtre est trop petite.

# À toi, avec aide

Une application mobile affiche l'écran d'accueil en effectuant ces appels vers un serveur à **RTT = 120 ms** (connexion HTTPS neuve pour le premier, réutilisée ensuite, bande passante négligeable) :

1. `GET /profil` ;
2. puis, après la réponse, `GET /cours?membre=...` ;
3. puis, pour chacun des **12 cours** de la liste, `GET /cours/{id}` en série ;
4. puis `GET /notifications`.

Calcule le temps total, puis propose et chiffre une version améliorée avec : appels indépendants en parallèle, remplacement de la boucle de 12 par une requête groupée, et réutilisation de la connexion. Combien d'allers-retours séquentiels reste-t-il sur le chemin critique ?

À toi, avec aide · réponse

# Indices progressifs

## Indice 1

La poignée de main de la première requête coûte 3 RTT en plus de son aller-retour.

## Indice 2

`/notifications` ne dépend pas des autres appels.

## Indice 3

Après regroupement, `/cours?...&details=1` renvoie tout.

À toi, avec aide · réponse

# Réponse

```
RTT = 0.120
poignee = 3 * RTT                                          # DNS + TCP + TLS (un
   e fois)
avant = poignee + (1 + 1 + 12 + 1) * RTT                   # profil, liste, 12 d
   étails, notifications : tout en série
print(f"avant : {avant * 1000:.0f} ms ({1 + 1 + 12 + 1} allers-retours en série)")
# après : notifications en parallèle avec profil ; liste avec détails (requête gro
   upée) ; connexion réutilisée
apres = poignee + 2 * RTT                                  # chemin critique : p
   rofil → liste groupée (notifications en parallèle)
print(f"après : {apres * 1000:.0f} ms (2 allers-retours sur le chemin critique)")
assert round(avant * 1000) == 2160 and round(apres * 1000) == 600
```

À toi, avec aide · réponse

# Réponse (suite)

**Avant : 2 160 ms** (3 RTT de poignée de main + 15 allers-retours en série). **Après : 600 ms** : `profil` puis `cours` avec détails (2 allers-retours sur le chemin critique), `notifications` partant en parallèle dès le début. Il reste **2 allers-retours séquentiels** (le second dépend de l'identifiant du membre obtenu par le premier). Pour descendre plus bas : mettre le profil dans le jeton d'authentification ou faire une requête d'agrégation unique côté serveur (BFF).

# À toi, en solo

**Agrégateur de requêtes (« data loader »).** Écris une classe `Chargeur` qui regroupe automatiquement les demandes individuelles :

1. `charger(cle)` renvoie un résultat futur ; les demandes arrivant dans une même « fenêtre » (même vague d'exécution) sont **regroupées** en un seul appel `charger_lot(cles)` ;
2. déduplique les clés identiques ;
3. mémorise les résultats déjà obtenus (cache) ;
4. simule une source distante avec **horloge virtuelle** : chaque appel coûte 80 ms ;
5. compare le nombre d'appels et le temps total pour afficher 30 cours (dont 10 en double) avec et sans chargeur ;
6. teste avec `asyncio` : 30 tâches concurrentes appelant `charger`.

À toi, en solo · réponse

# Pistes de correction

```
import asyncio

class Source:
    def __init__(self, rtt=0.080):
        self.rtt, self.appels, self.temps_virtuel = rtt, 0, 0.0
    def charger_lot(self, cles):
        self.appels += 1; self.temps_virtuel += self.rtt
        return {c: f"cours-{c}" for c in cles}

class Chargeur:
    def __init__(self, source):
        self.source, self.cache, self.attente, self.programme = source, {}, {}, Fa
   lse
        self.lots = []
    def charger(self, cle):
        boucle = asyncio.get_running_loop()
        if cle in self.cache:
            f = boucle.create_future(); f.set_result(self.cache[cle]); return f
        if cle not in self.attente:                                  # dédupli
```

À toi, en solo · réponse

# Pistes de correction (suite)

```
   cation
            self.attente[cle] = boucle.create_future()
        if not self.programme:                                  # on exé
   cutera le lot à la prochaine itération de la boucle
            self.programme = True
            boucle.call_soon(self._executer)
        return self.attente[cle]
    def _executer(self):
        lot, self.attente, self.programme = self.attente, {}, False
        self.lots.append(sorted(lot))
        resultats = self.source.charger_lot(list(lot))
        for cle, futur in lot.items():
            self.cache[cle] = resultats[cle]; futur.set_result(resultats[cle])

async def sans_chargeur(source, cles):
    for c in cles:                                              # N appe
   ls successifs
        source.charger_lot([c])
```

À toi, en solo · réponse

# Pistes de correction (suite)

```
async def avec_chargeur(source, cles):
    ch = Chargeur(source)
    res = await asyncio.gather(*(ch.charger(c) for c in cles))
    return res, ch

CLES = [i % 20 for i in range(30)]                                  # 30 dem
   andes dont 10 doublons
s1 = Source(); asyncio.run(sans_chargeur(s1, CLES))
s2 = Source(); res, ch = asyncio.run(avec_chargeur(s2, CLES))
print(f"sans chargeur : {s1.appels} appels, {s1.temps_virtuel * 1000:.0f} ms | ave
   c chargeur : {s2.appels} appel(s), {s2.temps_virtuel * 1000:.0f} ms")
assert s1.appels == 30 and s2.appels == 1 and len(ch.lots[0]) == 20
assert res == [f"cours-{c}" for c in CLES]                          # mêmes
   résultats, dans l'ordre
```

Le chargeur transforme **30 allers-retours** en **1**, en dédupliquant les 10 clés identiques. C'est le principe de `DataLoader` (GraphQL) et des clients de base de données par lots : on ne change pas le code appelant, on change la façon dont les demandes sont **regroupées**.

# Pièges classiques

| Piège | Pourquoi c'est un problème | Comment l'éviter |
|---|---|---|
| **Une requête par élément (N+1 réseau)** | N allers-retours en série | Requêtes groupées ou parallèles |
| **Nouvelle connexion à chaque appel** | Poignée de main répétée (3 RTT) | Keep-alive, pool de connexions, HTTP/2 |
| **Optimiser la bande passante quand la latence domine** | Aucun effet sur de petites requêtes | Compter les allers-retours sur le chemin critique |
| **Retransférer ce qui n'a pas changé** | Octets et temps gaspillés | ETag/304, Cache-Control, compression |

Mini-boss

# Mini-boss

**Cascade réseau de la page d'accueil de HashBox.** On te décrit la page : 1 document HTML, 3 feuilles de style, 8 scripts, 20 images, 4 appels d'API dont deux dépendent du résultat du premier. Latence 100 ms, bande passante 20 Mbit/s, tailles réalistes.

1. modélise la **cascade** (dépendances) et calcule le temps de chargement en tenant compte des connexions (6 en parallèle, HTTP/1.1) ;
2. propose 6 optimisations (HTTP/2, regroupement des scripts, compression, cache avec ETag, images adaptées et chargement différé, appels d'API groupés) et chiffre chacune avec ton modèle ;
3. identifie le **chemin critique** avant/après ;
4. vérifie par un petit serveur local les effets mesurables sans réseau réel (nombre de connexions, octets, 304) ;
5. écris un tableau gain/effort et recommande l'ordre ;
6. dessine la cascade avant/après en diagramme texte.

Mini-boss · réponse

# Pistes de correction

Un bon travail : modèle avec `max(chemin)` sur le graphe de dépendances (le HTML d'abord, puis CSS/scripts en parallèle limité à 6 connexions, puis API dépendantes), 6 optimisations chiffrées individuellement et en combinaison (les gains ne s'additionnent pas : Amdahl), vérification par compteurs sur serveur local, et un tableau qui place en tête les actions « peu coûteuses, gros effet » (réutilisation de connexions, compression, regroupement des appels API dépendants).

Quiz éclair

# Quiz éclair

**1.** Pour de petites requêtes lointaines, ce qui domine le temps est :

A. la bande passante
B. la latence (nombre d'allers-retours)
C. la couleur de la page

**2.** Une réponse `304 Not Modified` signifie :

A. la ressource est introuvable
B. une erreur du serveur
C. la ressource n'a pas changé, aucun corps n'est renvoyé

**3.** Le keep-alive permet de :

A. réutiliser une connexion pour plusieurs requêtes

Quiz éclair

# Quiz éclair (suite)

B. chiffrer les échanges
C. compresser les images

**4.** 🔁 *Révision mission 05* : Pour accélérer l'écriture d'un gros volume de lignes, on :

A. appelle flush après chaque ligne
B. regroupe les écritures dans un tampon
C. écrit octet par octet

**5.** 🔁 *Révision mission 05* : Une écriture atomique de fichier se fait en :

A. écrivant directement sur place
B. désactivant le disque
C. écrivant dans un fichier temporaire puis en le renommant

Quiz éclair · réponse

# Réponses

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

Je réfléchis

# Je réfléchis

1. Combien d'allers-retours séquentiels ta page ou ton API principale fait-elle avant d'afficher quelque chose d'utile ?
2. Qu'est-ce qui, dans ton application, est retransféré inutilement à chaque visite ?

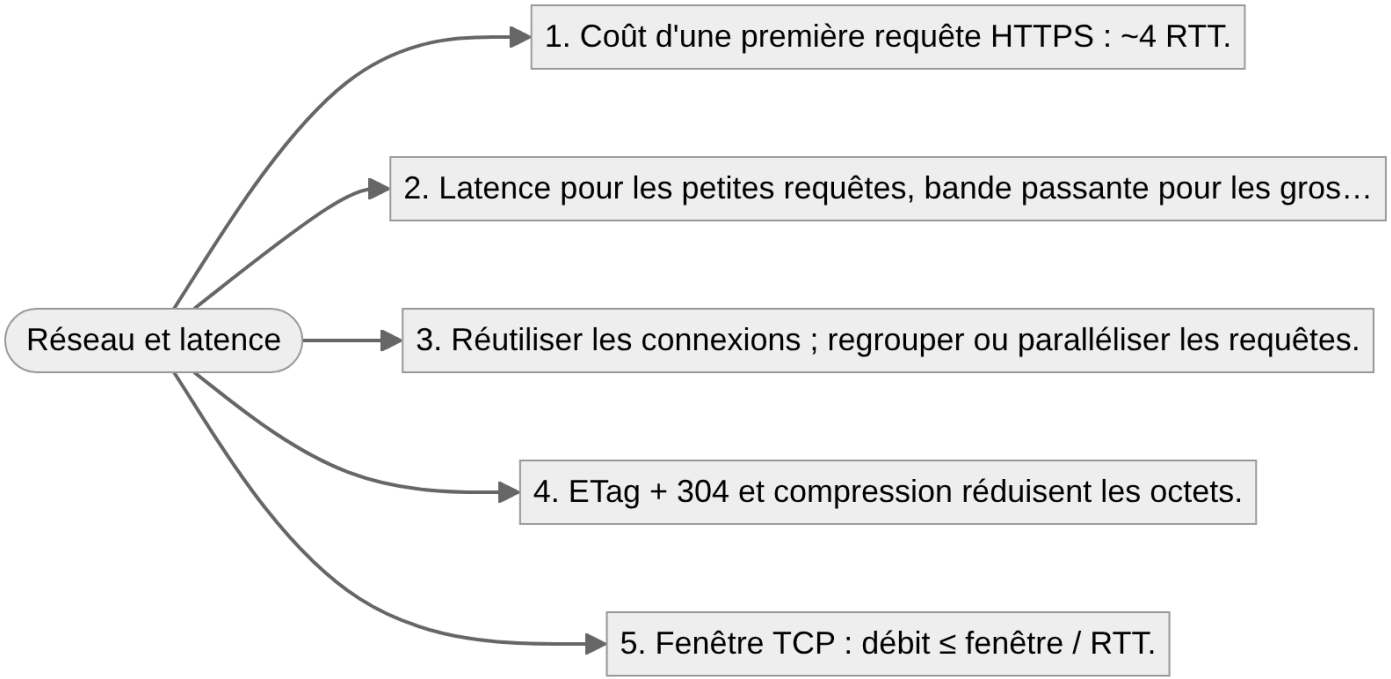
À retenir

# À retenir

1. Coût d'une première requête HTTPS : ~4 RTT.
2. Latence pour les petites requêtes, bande passante pour les gros transferts.
3. Réutiliser les connexions ; regrouper ou paralléliser les requêtes.
4. ETag + 304 et compression réduisent les octets.
5. Fenêtre TCP : débit ≤ fenêtre / RTT.

À retenir

# Le schéma de la mission



MISSION ACCOMPLIE

# +100 XP

## Badge débloqué : Dompteur de distances

Prochaine mission : Bases de données et performance

