Engineer

Temporisation exponentielle

⏳ attente avant nouvelle tentative multipliée à chaque échec

Ce que c’est

La temporisation exponentielle, ou exponential backoff, définit le temps d’attente avant de réessayer. Après chaque échec, l’attente est multipliée par un facteur fixe, le plus souvent deux, jusqu’à un plafond : une seconde, puis deux, quatre, huit, etc. La règle se trouve dans le client, mais elle est faite pour le système qu’il appelle. Un serveur, un chemin réseau ou une base qui cède sous la charge ne se rétablit pas plus vite parce qu’on l’interroge immédiatement de nouveau. Il se rétablit lorsque ses appelants ralentissent. Une attente croissante les fait ralentir davantage à mesure que le problème dure.

L’exemple le plus ancien et le plus répandu se trouve dans TCP. La RFC 6298, document du parcours de normalisation de juin 2011 qui définit le calcul du délai de retransmission TCP, fixe ce délai à une seconde avant la première mesure du temps aller-retour. Elle indique qu’une valeur calculée inférieure à une seconde devrait être arrondie à une seconde. Chaque fois que le temporisateur expire et qu’un segment doit être renvoyé, l’hôte doit doubler ce délai. Un plafond est permis s’il est d’au moins 60 secondes. À réception d’une nouvelle mesure aller-retour, le délai est recalculé depuis les mesures, ce qui peut le faire redescendre directement après plusieurs doublements. L’algorithme remonte à l’article Congestion Avoidance and Control de Van Jacobson, en 1988. La section sécurité de la RFC en expose l’enjeu : Internet dépend d’un temporisateur de retransmission correct pour garder le réseau stable et éviter son effondrement par congestion.

Dans le code applicatif, la même idée comporte quelques réglages : attente initiale, multiplicateur, attente maximale arrêtant la croissance et limite du nombre de tentatives ou du temps total consacré à réessayer. Après le n-ième échec, en comptant depuis zéro, l’attente est la plus petite valeur entre le plafond et base × multiplicateur à la puissance n. L’analyse AWS présentée plus bas appelle cela la temporisation exponentielle plafonnée. Le plafond compte

plus qu'il n'y paraît. Doubler dix fois une attente d'une seconde donne 1 024 secondes, plus de 17 minutes, au-delà de ce que la plupart des appelants attendront, quelle que soit l'opération.

Le dernier réglage est le jitter, un aléa dans chaque attente. Le protocole de temporisation des connexions gRPC décrit comment ses clients se reconnectent après un échec. Il part d'une seconde, multiplie par 1,6, plafonne à 120 secondes et applique un jitter de 0,2 : chaque attente est donc augmentée ou réduite aléatoirement, dans une limite de 20 %. Il laisse aussi au moins 20 secondes à chaque tentative de connexion pour aboutir. Il réinitialise la temporisation dès que le serveur a accepté la connexion, ce qu'il situe à la réception de la trame SETTINGS. Le protocole autorise d'autres implémentations, par exemple sur un téléphone devant se réveiller le moins souvent possible. Il exige toutefois de répartir les tentatives simultanées et d'éviter une fréquence sensiblement plus élevée.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que la temporisation exponentielle ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Une supervision et des alertes qui signalent les incidents tant qu'ils restent mineurs, avec les chiffres qui montrent à quelle fréquence ils le restent.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Premier tableau de bord d'observabilité — première vue de l'entreprise sur ses propres systèmes

Premier tableau de bord d'observabilité Contactez-nous →

Pourquoi c'est important

Une nouvelle tentative ajoute de la charge au pire moment : précisément lorsque le système appelé échoue. Si un service ralentit par surcharge et que chaque appelant réessaie immédiatement, la charge augmente pendant que la capacité baisse. La documentation de Google Cloud Storage classe pour cette raison les tentatives sans temporisation parmi les mauvaises pratiques : des reprises immédiates ou très rapprochées peuvent provoquer des défaillances en cascade, l'une en entraînant d'autres. La temporisation fait de la nouvelle tentative un frein plutôt qu'un amplificateur. Plus le problème dure, moins chaque client y ajoute de charge.

La temporisation seule ne suffit pas : les clients qui échouent ensemble attendent aussi ensemble. Dans Exponential Backoff And Jitter, publié par AWS en mars 2015, Marc Brooker a simulé de nombreux clients mettant à jour la même ligne d'une base distante avec un contrôle de concurrence optimiste. Quand chaque client participe à chaque tour, le travail total croît avec le carré du nombre de clients. La temporisation exponentielle plafonnée aide peu. Les appels restent groupés, séparés par des intervalles calmes ; la contention diminue à peine. Tirer chaque attente au hasard entre zéro et la valeur exponentielle plafonnée supprime ces intervalles, laisse un débit d'appels approximativement constant et, pour 100 clients concurrents, réduit le nombre d'appels de plus de moitié. Parmi les variantes comparées, Brooker qualifie celle sans jitter de « grande perdante ». Il en précise aussi la limite : aucune méthode ne change le fait que le travail croît avec le carré de la contention. L'aléa réduit le gaspillage, sans supprimer le conflit. Une mise à jour de mai 2023 indique que la plupart des SDK AWS incluent désormais temporisation exponentielle et jitter dans leurs modes de reprise standard et adaptatif.

La temporisation décide quand réessayer, pas si cela est sûr. La documentation Cloud Storage pose d'abord deux questions. La réponse indique-t-elle un problème transitoire : codes HTTP 408, 429 et 5xx, dépassement du délai d'attente d'un socket ou déconnexion TCP ? La requête est-elle idempotente ? Une opération de liste peut être répétée librement. La création d'une notification Pub/Sub ne le peut pas : chaque appel réussi crée une nouvelle notification. Entre les deux, les opérations conditionnellement idempotentes deviennent sûres à relancer si elles portent une précondition. Un téléversement peut ainsi ne réussir que si le numéro de génération de l'objet correspond encore à celui attendu par le client.

Le serveur en sait parfois plus que la formule du client. L'en-tête Retry-After, défini par la RFC 9110, indique combien de temps le client devrait attendre avant une requête suivante, sous forme de date ou de nombre de secondes. Avec une réponse 503 Service Unavailable, il précise la durée d'indisponibilité prévue. Le statut 429 Too Many Requests de la RFC 6585, qui signale une limitation de débit, peut porter le même champ. Une boucle de reprise peut employer cette valeur à la place de son calcul, puisqu'elle provient du côté qui connaît sa propre charge.

À quoi cela ressemble en pratique

Une boucle de reprise avec temporisation exponentielle plafonnée et jitter complet tient en quelques lignes de Python :

```
import random
import time

class TransientError(Exception):
    """A timeout, a 408, 429 or 5xx response, or a dropped connection."""

def call_with_backoff(op, attempts=5, base=1.0, cap=30.0):
    for attempt in range(attempts):
        try:
            return op()
        except TransientError:
            if attempt == attempts - 1:
                raise
            ceiling = min(cap, base * 2**attempt)
            time.sleep(random.uniform(0, ceiling))
```

Avec ces valeurs, cinq tentatives sont séparées par quatre attentes plafonnées à 1, 2, 4 et 8 secondes. La boucle passe donc au plus 15 secondes à dormir avant d'abandonner, environ 7,5 secondes en moyenne, chaque attente étant tirée uniformément entre zéro et son plafond. Seule `TransientError` entraîne une nouvelle tentative ; un échec d'autorisation ou une requête mal formée lève immédiatement une exception. La boucle répète tout ce que fait `op` . `op` doit donc pouvoir être répété sans risque : une lecture ou une écriture que le serveur peut reconnaître comme répétée.

Les bibliothèques clientes fournissent leurs propres variantes, avec des réglages prédéfinis différents, même pour un service identique. Pour Cloud Storage, le client Python part d'une seconde, double chaque attente jusqu'à 60 secondes et abandonne après un délai prédéfini de 120 secondes. Le client C++ part aussi d'une seconde et double, mais peut attendre jusqu'à cinq minutes entre deux tentatives et réessayer pendant au plus 15 minutes, plus de sept fois plus longtemps. Il relance aussi des opérations jamais idempotentes, sauf si le programme choisit explicitement une politique stricte d'idempotence. La ligne de commande gcloud emploie ce que Google appelle une temporisation exponentielle binaire tronquée, avec au plus 32 nouvelles tentatives sans autre réglage. Deux programmes appelant la même API face au même échec peuvent donc le signaler après deux minutes ou un quart d'heure, selon leur langage.

L'attente n'a pas à bloquer un thread. En arrière-plan, une tâche échouée peut retourner dans une file avec une date d'exécution ultérieure. La temporisation devient alors une propriété de la tâche plutôt que d'un processus endormi. C'est sa place naturelle lorsque personne n'attend la réponse.

Dans le projet courriel avec basculement et journalisation de la livraison, réalisé dans le cadre d'un poste occupé depuis 2025, la temporisation se trouve du côté asynchrone. Les messages passent par trois fournisseurs : SendGrid en premier, puis SMTP2GO et Azure Communication Services, avec basculement automatique. L'envoi lui-même utilise un client au délai d'expiration de 30 secondes. La livraison asynchrone dispose de nouvelles tentatives et d'une temporisation, afin qu'un bref incident ne coûte pas un message. La page résume le résultat : si un fournisseur traverse un mauvais après-midi, le courrier suit un chemin dégradé plutôt que de subir une panne. Elle nomme aussi le prix : trois intégrations à maintenir au lieu d'une.

Où les problèmes surviennent

Tous les clients attendent la même durée. Sans jitter, ceux qui ont échoué ensemble réessaient ensemble. Le serveur retrouve le même pic après une, deux, quatre et huit secondes. La simulation de Brooker montre que la temporisation exponentielle seule réduit à peine la contention pour cette raison. Ajoutez un aléa : dans sa comparaison, Full Jitter et Decorrelated Jitter réduisent tous deux nettement le travail client et la charge serveur. Les deux sont de bons choix initiaux.

L'attente n'a pas de plafond, ou les tentatives n'ont pas de fin. Un doublement non plafonné dépasse 17 minutes après dix échecs. Une boucle sans limite garde la requête active alors que plus personne n'attend sa réponse depuis longtemps. Google classe le non-respect des limites de reprise parmi les mauvaises pratiques : réessayer indéfiniment peut épuiser les ressources applicatives ou continuer à solliciter un service qui ne se rétablira pas sans intervention. Fixez une attente maximale et un délai total adaptés à l'appelant : courts si la latence compte, plus longs pour le traitement par lots.

On réessaie malgré des erreurs permanentes. Une erreur d'autorisation ou une requête invalide produit le même échec à chaque tentative. Réessayer ne fait que retarder l'erreur. Sans limite, le client peut finir dans une boucle infinie, avertit Cloud Storage. Distinguez les erreurs transitoires des permanentes avant la boucle. Ne réessayez que les premières, comme 408, 429, 5xx et les connexions interrompues. Journalisez les autres avec leur cause.

Une opération non idempotente est répétée aveuglément. Si la requête expire après que le serveur a agi, la relance répète l'action. Google cite parmi ses mauvaises pratiques les écrasements ou suppressions involontaires de données

qui en résultent. Ne relancez que les opérations idempotentes, rendez les écritures conditionnelles quand l'API le permet et, sinon, assurez-vous que le destinataire détecte les doublons.

Plusieurs couches empilent leurs tentatives. Si l'application effectue trois tentatives et que sa bibliothèque cliente en effectue trois pour chacune, un échec aboutit à neuf tentatives. C'est l'exemple de Google. Chaque couche supplémentaire qui réessaie seule multiplie encore ce nombre. La documentation ajoute que ces reprises peuvent entraîner une limitation des requêtes et une latence accrue sans améliorer le taux de succès. Réessayez dans une seule couche, généralement la bibliothèque, et désactivez ou réduisez les reprises ailleurs.

La temporisation n'est jamais réinitialisée, ou personne ne la voit. Un client qui conserve sa dernière longue attente après le rétablissement attendra le maximum au petit échec suivant. gRPC revient à l'attente initiale dès qu'une connexion est acceptée, et TCP recalcule son temporisateur depuis de nouvelles mesures. Réinitialisez après un succès. Comme les reprises cachent les problèmes aux utilisateurs, elles les cachent aussi à l'équipe. Comptez-les dans la supervision et déclenchez une alerte si leur fréquence grimpe : une dépendance exigeant trois tentatives par appel présente déjà des défaillances.

Pour aller plus loin

- RFC 6298 : calcul du temporisateur de retransmission TCP — les règles du parcours de normalisation pour le délai de retransmission TCP, dont son doublement à chaque expiration.
- Protocole de temporisation des connexions gRPC — l'algorithme de reconnexion, ses cinq paramètres et leurs valeurs, et le moment de la réinitialisation.
- Temporisation exponentielle et jitter — la simulation de clients concurrents par Marc Brooker en 2015, comparant l'absence de jitter aux variantes Full, Equal et Decorrelated Jitter.
- Stratégie de nouvelle tentative — les opérations que Cloud Storage juge sûres à relancer, les réglages prédéfinis de chaque bibliothèque et cinq mauvaises pratiques de reprise.
- RFC 9110 : sémantique HTTP — le champ Retry-After, par lequel le serveur indique l'attente avant la prochaine requête.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le parcours de déploiement, les risques opérationnels et les personnes d'astreinte. Nous pouvons définir un changement avec des vérifications, une procédure de reprise et une responsabilité claire.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que la temporisation exponentielle ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/exponential-backoff/