Délais d'expiration, nouvelles tentatives et interruption avec instabilité

Marc Brooker

Délais d'expiration, nouvelles tentatives et interruption avec instabilité

Les échecs se produisent parfois

Chaque fois qu'un service ou un système en appelle un autre, des défaillances peuvent survenir. Ces échecs peuvent provenir de nombreux facteurs. Ils comprennent des serveurs, des réseaux, des équilibreurs de charge, des logiciels, des systèmes d'exploitation ou même des erreurs commises par des opérateurs système. Nous concevons nos systèmes de manière à réduire le risque de défaillance, mais nous ne pouvons pas construire de systèmes infaillibles. Ainsi, dans Amazon, nous concevons nos systèmes de manière à tolérer et à réduire le risque de défaillance, et à éviter d'aggraver un faible pourcentage de défaillances pour les voir se transformer en panne générale. Pour créer des systèmes résilients, nous utilisons trois outils essentiels : les délais d'expiration, les nouvelles tentatives et l'interruption.

De nombreux types d'échecs deviennent apparents lorsque les demandes prennent plus de temps que d'habitude, voire ne sont jamais terminées. Lorsqu'un client doit attendre plus que d'habitude qu'une demande soit terminée, il monopolise également les ressources qu'il utilisait pour cette demande plus longtemps. Lorsqu'un certain nombre de demandes conservent des ressources pendant une longue période, le serveur peut manquer de ces ressources. Elles peuvent inclure de la mémoire, des threads, des connexions, des ports éphémères ou tout autre élément limité. Pour éviter d'en arriver là, les clients définissent des *délais d'expiration*. Les délais d'expiration sont la durée maximale pendant laquelle un client attend la fin d'une demande.

Réessayer l'opération avec la même demande aboutit souvent. Cela s'explique par le fait que les types de systèmes que nous créons n'échouent pas souvent en tant qu'unité unique. Ils souffrent plutôt d'échecs partiels ou transitoires. Un échec partiel survient lorsqu'un pourcentage de demandes aboutit. Un échec transitoire survient lorsqu'une demande échoue pendant une courte période. Les *nouvelles tentatives* permettent aux clients de survivre à ces échecs partiels aléatoires et échecs transitoires de courte durée en renvoyant la même demande.

Ces nouvelles tentatives ne sont pas toujours prudentes. Une nouvelle tentative peut augmenter la charge sur le système appelé, si le système échoue déjà parce qu'il approche d'une surcharge. Pour éviter ce problème, nous implémentons nos clients pour utiliser l'*interruption*. Cela augmente le temps entre les tentatives suivantes, ce qui maintient la charge sur le serveur même. L'autre problème avec les tentatives est que certains appels distants génèrent des effets secondaires. Un délai d'expiration ou un échec ne signifie pas nécessairement que les effets secondaires ne sont pas survenus. Si vous ne souhaitez pas voir les effets secondaires apparaître à plusieurs reprises, nous vous conseillons de concevoir des API idempotentes, ce qui signifie qu'elles peuvent être réessayées en toute sécurité.

Enfin, le trafic n'arrive pas dans les services Amazon à un débit constant. Au lieu de cela, la fréquence d'arrivée des demandes connaît souvent d'importantes rafales. Ces rafales peuvent être causées par le comportement du client, la reprise après sinistre et même par un élément simple comme une tâche cron périodique. Si des erreurs sont provoquées par la charge, les tentatives peuvent être inefficaces si tous les clients réessayent en même temps. Pour éviter ce problème, nous utilisons l'*instabilité*. Il s'agit d'un délai aléatoire avant de formuler ou de réessayer une demande afin d'empêcher d'importantes rafales en répartissant la fréquence d'arrivée.

Chacune de ces solutions est abordée dans les sections suivantes.

Délais d'expiration

Une des meilleures pratiques d'Amazon consiste à définir un délai d'expiration pour tout appel distant, et généralement pour tout appel entre processus, même sur la même boîte. Cela inclut à la fois un délai de connexion et un délai de demande. De nombreux clients standard offrent des fonctionnalités robustes de délai d'expiration intégrées.

Généralement, le problème le plus difficile consiste à choisir une valeur de délai d'expiration à définir. Définir un délai d'expiration trop élevé réduit son utilité, car les ressources sont toujours consommées tant que le client attend l'expiration. Définir un délai d'expiration trop faible présente deux risques :

- Augmentation du trafic sur le serveur et de la latence, en raison d'un nombre trop élevé de nouvelles tentatives de demandes.
- Augmentation de la faible latence du serveur conduisant à une panne totale, car les nouvelles tentatives de toutes les demandes sont lancées à la fois.

Pour choisir un délai d'expiration pour les appels au sein d'une région AWS, il est recommandé de commencer par les métriques de latence du service en aval. Ainsi, chez Amazon, lorsque nous faisons en sorte qu'un service en appelle un autre, nous choisissons un taux acceptable de faux délais d'expiration (tels que 0,1 %). Ensuite, nous examinons le centile de latence correspondant sur le service en aval (p99.9 dans cet exemple). Cette approche fonctionne bien dans la plupart des cas, mais présente quelques pièges, tels que les suivants :

- Cette approche ne fonctionne pas dans les cas où les clients ont une latence réseau importante, par exemple, via Internet. Dans ces cas, nous tenons compte de la latence réseau raisonnable dans le pire des cas, en gardant à l'esprit que les clients peuvent couvrir le monde entier.

- Cette approche ne fonctionne pas non plus avec les services qui ont des limites de latence serrées, où p99.9 est proche de p50. Dans ces cas, l'ajout d'un certain remplissage nous aide à éviter de petites augmentations de latence qui entraînent un grand nombre de délais d'expiration.

- Nous avons rencontré un piège commun lors de l'implémentation des délais d'expiration. SO_RCVTIMEO de Linux est puissant, mais présente des inconvénients qui le rendent inutilisable en tant que délai d'expiration de socket de bout en bout. Certains langages, tels que Java, exposent directement ce contrôle. D'autres langages, tels que Go, fournissent des mécanismes de délai d'expiration plus robustes.

- Il existe également des implémentations dans lesquelles le délai d'expiration ne couvre pas tous les appels distants, tels que les liaisons DNS ou TLS. En général, nous préférons utiliser les délais d'expiration intégrés à des clients bien testés. Si nous implémentons nos propres délais d'expiration, nous portons une attention particulière à la signification exacte des options de socket de délai d'expiration et aux travaux en cours.

Dans un système sur lequel j'ai travaillé chez Amazon, nous avons vu un petit nombre de délais d'expiration parler à une dépendance immédiatement après les déploiements. Le délai

d'expiration était très faible, à environ 20 millisecondes. En dehors des déploiements, même avec cette valeur de délai d'expiration faible, nous n'avons pas constaté de délais d'expiration réguliers. En creusant, j'ai découvert que le minuteur incluait l'établissement d'une nouvelle connexion sécurisée, qui était réutilisée lors de demandes ultérieures. Étant donné que l'établissement de la connexion a pris plus de 20 millisecondes, nous avons observé un petit nombre de demandes expirer lorsqu'un nouveau serveur est entré en service après des déploiements. Dans certains cas, les demandes ont été réessayées et ont abouti. Nous avons initialement résolu ce problème en augmentant la valeur du délai d'expiration au cas où une connexion serait établie. Par la suite, nous avons amélioré le système en établissant ces connexions lors du démarrage d'un processus, mais avant de recevoir du trafic. Cela nous a permis de contourner le problème du délai d'expiration.

Nouvelles tentatives et interruption

Les nouvelles tentatives sont « égoïstes ». En d'autres termes, lorsqu'un client effectue une nouvelle tentative, il passe plus de temps sur le serveur pour augmenter ses chances de réussite. Quant aux échecs rares ou transitoires, ils ne sont pas un problème. En effet, le nombre total de nouvelles demandes est faible et le compromis de la disponibilité apparente croissante fonctionne bien. Lorsque les échecs sont provoqués par une surcharge, les nouvelles tentatives qui augmentent la charge peuvent aggraver considérablement les choses. Elles peuvent même retarder la récupération en maintenant la charge élevée longtemps après la résolution du problème initial. Les tentatives sont similaires à un médicament puissant : elles sont utiles à la bonne dose, mais peuvent causer des dommages importants en cas d'utilisation excessive. Malheureusement, dans les systèmes distribués, il n'y a pratiquement aucun moyen de coordonner tous les clients pour obtenir le nombre approprié de tentatives.

La solution que nous préférons utiliser dans Amazon est l'interruption. Au lieu de réessayer immédiatement et de manière agressive, le client attend un certain délai entre les différentes tentatives. Le modèle le plus courant est une *interruption exponentielle*, dans lequel le temps d'expiration augmente de manière exponentielle après chaque tentative. L'interruption exponentielle peut entraîner des délais d'interruption très longs, car les fonctions exponentielles se développent rapidement. Pour éviter que les nouvelles tentatives ne durent trop longtemps, les implémentations limitent généralement leur interruption à une valeur maximale. Ceci s'appelle, sans surprise, une *interruption exponentielle plafonnée*. Toutefois, cela implique un autre problème. Désormais, tous les clients réessayent constamment au taux plafonné. Dans presque tous les cas, notre solution consiste à limiter le nombre de tentatives du client et à gérer la défaillance qui en résulte plus tôt dans l'architecture orientée service. Dans la plupart des cas, le client va quand même abandonner l'appel, car il a ses propres délais.

Il existe d'autres problèmes liés aux tentatives, tels que les suivants :

- Les systèmes distribués ont souvent plusieurs couches. Prenons un système dans lequel l'appel du client génère une pile d'appels de service de cinq couches. Il se termine par une demande sur une base de données et trois tentatives à chaque couche. Que se passe-t-il lorsque la base de données commence à faire échouer des demandes en cours de chargement ? Si chaque couche tente à nouveau indépendamment, la charge de la base de données augmentera de 243 fois, ce qui la

rendra impossible à récupérer. En effet, les nouvelles tentatives de chaque couche se multiplient : trois premières tentatives, puis neuf tentatives, etc. Au contraire, réessayer au niveau de la couche la plus haute de la pile peut entraîner une perte de travail des appels précédents, ce qui réduit l'efficacité. En règle générale, pour les opérations dans le plan de contrôle et dans le plan de données à faible coût, nous recommandons de réessayer à un seul point de la pile.

- La charge. Même avec une seule couche d'essais, le trafic augmente encore considérablement lorsque des erreurs commencent à apparaître. Les *disjoncteurs*, où les appels vers un service en aval sont complètement arrêtés lorsqu'un seuil d'erreur est dépassé, sont largement recommandés pour résoudre ce problème. Malheureusement, les disjoncteurs introduisent un comportement modal dans les systèmes qui peuvent être difficiles à tester et peuvent introduire un temps d'ajout important à la reprise. Nous avons constaté que nous pouvions atténuer ce risque en limitant les nouvelles tentatives au niveau local à l'aide d'un compartiment à jetons. Cela permet à tous les appels de réessayer tant qu'il y a des jetons, puis de réessayer à un taux fixe lorsque les jetons sont épuisés. AWS a ajouté ce comportement au kit SDK AWS en 2016. Ainsi, les clients qui utilisent le kit SDK ont ce comportement de limitation intégré.

- Décider quand réessayer. En règle générale, nous pensons que les API comportant des effets indésirables ne peuvent être essayées à nouveau, sauf si elles fournissent une idempotence. Cela garantit que les effets secondaires ne se produisent qu'une seule fois, quelle que soit la fréquence à laquelle vous réessayez. Les API en lecture seule sont généralement idempotentes, contrairement aux API de création de ressources. Certaines API, telles que l'API RunInstances Amazon Elastic Compute Cloud (Amazon EC2), fournissent des mécanismes explicites basés sur des jetons pour fournir l'idempotence et assurer leur sécurité pour de nouvelles tentatives. Une bonne conception des API et des précautions lors de la mise en œuvre des clients sont nécessaires pour éviter les effets secondaires en double.

- Savoir quels échecs valent la peine d'essayer à nouveau. HTTP établit une distinction claire entre les erreurs *client* et *serveur*. Cela indique que les erreurs client ne doivent pas être répétées avec la même demande, car elles ne réussiront pas plus tard, alors que les erreurs du serveur peuvent réussir lors des tentatives suivantes. Malheureusement, la cohérence éventuelle des systèmes brouille considérablement cette ligne. Une erreur du client à un moment donné peut se transformer en succès au moment suivant, lorsque l'état se propage.

En dépit de ces risques et de ces difficultés, les nouvelles tentatives sont un mécanisme puissant pour assurer une haute disponibilité face aux erreurs temporaires et aléatoires. Le jugement est nécessaire pour trouver le bon compromis pour chaque service. Selon notre expérience, un bon point de départ est de rappeler que les nouvelles tentatives sont égoïstes. Les nouvelles tentatives sont un moyen pour les clients d'affirmer l'importance de leur demande et d'exiger que le service dépense davantage de ressources pour la traiter. Si un client est trop égoïste, il peut créer de nombreux problèmes.

Instabilité

Lorsque les échecs sont causés par une surcharge ou des conflits, il est souvent difficile de prendre du recul. Ceci est dû à la corrélation. Si tous les appels en échec sont rétablis à la même heure, ils provoquent à nouveau des conflits ou une surcharge lorsqu'ils sont réessayés. L'instabilité est notre solution. L'instabilité ajoute un peu d'aléatoire à l'interruption afin de répartir les nouvelles tentatives dans le temps. Pour plus d'informations sur la quantité d'instabilité à ajouter et sur les meilleures façons de l'ajouter, consultez la section Interruption exponentielle et instabilité.

L'instabilité n'est pas seulement réservée aux nouvelles tentatives. L'expérience opérationnelle nous a appris que le trafic vers nos services, y compris les plans de contrôle et les plans de données, tend à augmenter considérablement. Ces pics de trafic peuvent être très courts et sont souvent masqués par des métriques agrégées. Lors de la construction de systèmes, nous envisageons d'ajouter un peu d'instabilité à tous les temporisateurs, travaux périodiques et autres travaux différés. Cela permet de répartir les pics de travail et facilite la mise à l'échelle des services en aval pour une charge de travail.

Lors de l'ajout de l'instabilité au travail planifié, nous ne sélectionnons pas l'instabilité sur chaque hôte de manière aléatoire. Au lieu de cela, nous utilisons une méthode cohérente qui produit le même nombre à chaque fois sur le même hôte. De cette manière, si un service est surchargé ou si une situation de concurrence critique survient, cela se produit de la même manière dans un modèle. Nous, les humains, sommes capables d'identifier les modèles et nous sommes plus susceptibles de déterminer la cause racine. L'utilisation d'une méthode aléatoire garantit que si une ressource est surchargée, cela ne se produit que de manière aléatoire. Cela rend le dépannage beaucoup plus difficile.

Sur les systèmes sur lesquels j'ai travaillé, tels que Amazon Elastic Block Store (Amazon EBS) et AWS Lambda, nous avons constaté que les clients envoient fréquemment des demandes à intervalles réguliers, par exemple, une fois par minute. Toutefois, lorsqu'un client a plusieurs serveurs se comportant de la même manière, ils peuvent s'aligner et déclencher leurs demandes en même temps. Il peut s'agir des premières secondes d'une minute ou des premières secondes après minuit pour les travaux quotidiens. En accordant une attention particulière à la charge par seconde et en travaillant avec les clients pour rendre leurs charges de travail périodiques instables, nous avons effectué la même quantité de travail avec une capacité de serveur inférieure.

Nous avons moins de contrôle sur les pointes du trafic client. Toutefois, même pour les tâches déclenchées par le client, il est judicieux d'ajouter une instabilité qui n'a aucune incidence sur l'expérience client.

Conclusion

Dans les systèmes distribués, les défaillances transitoires ou la latence dans les interactions à distance sont inévitables. Les délais d'expiration empêchent les systèmes de prendre trop de temps, les nouvelles tentatives peuvent masquer ces défaillances, et le recul et l'instabilité peuvent améliorer l'utilisation et réduire les encombrements des systèmes.

Chez Amazon, nous avons appris qu'il est important de faire preuve de prudence lors de nouvelles tentatives. Les nouvelles tentatives peuvent amplifier la charge sur un système dépendant. Si les appels vers un système arrivent à expiration et que ce système est surchargé, les nouvelles tentatives peuvent aggraver la surcharge au lieu de l'améliorer. Nous évitons cette amplification en réessayant uniquement lorsque nous observons que la dépendance est saine. Nous arrêtons les nouvelles tentatives lorsque celles-ci n'aident pas à améliorer la disponibilité.