Les caches distribués et Redis

Nizar GUERRABI / Mohamed amine SALMANE
Polytech grenoble

22/11/2025

Résumé

Les systèmes applicatifs modernes doivent répondre à des exigences fortes de performance, de scalabilité et de réactivité. La mise en cache permet de réduire la latence en évitant des accès répétés à des bases de données plus lentes. Cependant, le cache local atteint vite ses limites dès que l'on répartit l'application sur plusieurs serveurs. Les caches distribués, et en particulier Redis, fournissent un espace de stockage en mémoire centralisé, partagé par toutes les instances. Cette synthèse présente les concepts de cache, les différences entre cache local et distribué, les principaux patterns de cache (cache-aside, read-through, write-through, write-behind) et le rôle de Redis comme outil de référence pour la mise en place d'un cache distribué dans les architectures contemporaines.

Mots-clés

Cache, Cache distribué, Redis, Performance, Scalabilité, Systèmes distribués, Patterns de cache, Base de données.

Abstract

Modern software systems must meet strict performance, scalability, and responsiveness requirements. Caching reduces latency by avoiding repeated access to slower databases. However, local caches reach their limits as soon as the application is distributed across multiple servers. Distributed caches, especially Redis, provide an in-memory, centralized store shared by all instances. This paper introduces the notion of caching, compares local and distributed caches, reviews the main caching patterns (cache-aside, read-through, write-through, write-behind), and describes the role of Redis as a de facto standard for implementing distributed caching in modern architectures.

Keywords

Cache, Distributed cache, Redis, Performance, Scalability, Distributed systems, Caching patterns, Database.

1. Introduction : pourquoi la mise en cache ?

Les applications web et mobiles actuelles doivent répondre à un nombre croissant d'utilisateurs tout en offrant des temps de réponse très faibles. Accéder directement à une base de données relationnelle ou NoSQL pour chaque requête devient rapidement un goulot d'étranglement : la latence disque est bien supérieure à celle de la mémoire, et la base doit gérer un volume important de connexions concurrentes. La mise en cache consiste à stocker, dans une mémoire plus rapide (généralement la RAM), des données fréquemment consultées afin d'éviter des accès répétitifs à la base de données ou à des services externes.

Intuitivement, un cache joue le rôle de "mémoire à court terme" de l'application : quand une donnée est demandée, on va d'abord la chercher dans le cache ; si elle est présente, la réponse est quasi instantanée, sinon on consulte la source d'autorité (base de données) puis on alimente le cache pour les requêtes futures. Cette idée simple se décline ensuite en différents modèles de déploiement (local vs distribué) et en divers patterns de lecture et d'écriture.

2. Cache local et ses limites

Le *cache local* est la forme la plus simple de cache. Il est directement intégré au code applicatif de l'application : typiquement, une structure de type dictionnaire ou *HashMap* en mémoire vive. Lorsqu'une requête arrive, l'application vérifie si la donnée est présente dans cette structure en RAM ; si oui, elle la renvoie immédiatement. Sinon, elle interroge la base de données, puis stocke la réponse dans le cache local avant de la retourner.

Ce modèle offre plusieurs avantages évidents :

— latence très faible (accès direct en mémoire, sans réseau) ;
— aucune dépendance externe (pas de serveur supplémentaire à déployer) ;
— implémentation simple dans la plupart des langages.

Cependant, cette approche présente des limites importantes dès que l'on passe à une architecture avec plusieurs instances applicatives (scaling horizontal) :

— **Perte au redémarrage** : le cache vit dans la RAM du processus ; si le serveur ou le conteneur redémarre, tout le contenu du cache est perdu, ce qui provoque un pic de charge sur la base de données le temps de le reconstruire.
— **Caches multiples non synchronisés** : chaque instance d'application dispose de son propre cache en mémoire. Deux serveurs peuvent donc contenir des valeurs différentes pour la même clé, selon l'historique des requêtes qui leur sont arrivées. La cohérence globale devient difficile à garantir.
— **Invalidation complexe** : lorsqu'une donnée est modifiée dans la base de données, il faudrait invalider ou mettre à jour la valeur correspondante dans *tous* les caches locaux. Mettre en place cette synchronisation est complexe, surtout à grande échelle.
— **Duplication mémoire** : si chaque instance stocke les mêmes données populaires, on gaspille de la mémoire globale du système.

Ces limitations rendent le cache local peu adapté aux architectures distribuées modernes (microservices, conteneurs, orchestrateurs comme Kubernetes), où plusieurs instances sont déployées pour des raisons de scalabilité et de haute disponibilité.

3. Notion de cache distribué

Pour résoudre ces problèmes, on introduit la notion de *cache distribué*. Plutôt que de maintenir un cache séparé dans chaque instance applicative, on déploie un composant externe, généralement un serveur ou un cluster, qui fournit un cache centralisé partagé par toutes les instances. Ce composant offre une interface de type clé–valeur (*key–value store*) accessible via le réseau.

Schématiquement, toutes les instances d'application font leurs opérations de lecture et d'écriture sur un cache unique, typiquement un serveur Redis ou Memcached. Cela apporte plusieurs bénéfices :

- **Cohérence des données** : une mise à jour ou invalidation dans le cache est immédiatement visible pour toutes les instances.
- **Réduction de la duplication** : les données fréquemment accédées ne sont stockées qu'à un seul endroit.
- **Survie aux redémarrages applicatifs** : si une instance applicative tombe, le cache distribué reste disponible pour les autres.
- **Gestion avancée** : le composant de cache peut offrir du TTL (expiration automatique), des stratégies d'éviction, de la réplication, du clustering, etc.

Parmi les solutions de cache distribué, Redis s'est imposé comme une référence grâce à sa rapidité, la richesse de ses structures de données et ses capacités de persistance optionnelle.

4. Présentation de Redis

Redis (*REmote DIctionary Server*) est un *data store* en mémoire, orienté clé–valeur, classé dans la famille des bases NoSQL. Il est conçu à l'origine comme un cache, mais son modèle et ses fonctionnalités lui permettent aussi d'être utilisé comme base de données primaire dans certains scénarios. Redis stocke ses données principalement en RAM, ce qui lui permet d'offrir des temps de réponse de l'ordre de la microseconde.

Au-delà de la simple paire clé–valeur, Redis supporte plusieurs structures de données : chaînes de caractères, listes, ensembles, ensembles ordonnés, tables de hachage, hyperloglogs, flux (*streams*), etc. Cette variété facilite l'implémentation de cas d'usage comme la gestion de sessions, les compteurs en temps réel, les files de messages, ou encore la fonctionnalité de *pub/sub*. Redis propose également :

- un système de TTL (expiration) par clé ;
- des politiques d'éviction configurables (LRU, LFU, etc.) ;
- des mécanismes de persistance (snapshots RDB, fichiers AOF) ;
- la réplication maître–réplique ;
- le clustering pour répartir les données sur plusieurs nœuds.

Ces caractéristiques font de Redis un candidat naturel pour implémenter un cache distribué robuste et performant dans des architectures cloud ou microservices.

5. Patterns de cache : gestion des lectures et des écritures

L'utilisation d'un cache, qu'il soit local ou distribué, ne se limite pas à l'existence d'un stockage rapide. Il faut aussi définir *comment* l'application interagit avec le cache, notamment pour les opérations de lecture et d'écriture. Plusieurs patterns reconnus sont décrits dans la littérature et les guides d'architecture.

Cache-aside (lazy loading). C'est le pattern le plus courant avec Redis. Lorsqu'une application a besoin d'une donnée, elle commence par interroger le cache. Si la donnée est présente (*cache hit*), elle la renvoie directement. En cas d'absence (*cache miss*), l'application consulte la base de données, renvoie le résultat, puis l'insère dans le cache pour les prochains accès. Ce modèle place la responsabilité de la gestion du cache dans l'application, ce qui offre une grande flexibilité mais impose une logique explicite.

Read-through. Dans ce pattern, l'application ne parle qu'au cache. Si une entrée n'est pas trouvée, c'est la couche de cache elle-même (ou un proxy) qui interroge la base de données, met à jour le cache et renvoie la donnée. Cela simplifie le code applicatif, mais nécessite un composant de cache plus sophistiqué ou des bibliothèques spécifiques.

Write-through. Lors d'une écriture, l'application met à jour simultanément la base de données et le cache. Le cache reste toujours cohérent avec la source de vérité, au prix d'un coût d'écriture plus élevé (deux opérations au lieu d'une).

Write-behind (write-back). Ici, l'application écrit uniquement dans le cache, et le cache propage ensuite les modifications vers la base de données de manière asynchrone. Cette stratégie améliore les performances d'écriture mais introduit un risque de perte de données en cas de panne du cache avant la synchronisation, et une latence potentielle dans la cohérence avec la base.

Redis peut être utilisé dans ces différents patterns, même si, dans la pratique, le modèle cache-aside est de loin le plus répandu dans les applications qui utilisent Redis comme cache distribué.

6. Redis comme cache distribué dans une architecture multi-serveurs

Dans une architecture moderne, il est courant de placer un équilibreur de charge (*load balancer*) devant plusieurs instances d'un même service applicatif. Sans cache distribué, chaque instance devrait reconstruire son propre cache local, ce qui entraîne incohérences et surcharge inutile.

En introduisant Redis comme cache distribué, toutes les instances d'application se connectent au même serveur (ou cluster) Redis. Lorsqu'une donnée est écrite ou invalidée, l'effet est immédiatement visible pour toutes les instances. À la lecture, le cache peut absorber la majorité des requêtes, ne laissant à la base que les *cache misses* et les opérations d'écriture confirmées. Cette configuration permet :

— de réduire la charge sur la base de données ;
— d'améliorer la latence perçue par l'utilisateur final ;
— de simplifier la gestion de la cohérence du cache ;
— de scaler horizontalement les instances applicatives sans multiplier les caches locaux.

Comme le rappellent plusieurs comparatifs Redis vs Memcached, Redis est particulièrement apprécié dans ce rôle grâce à ses structures de données avancées, et son écosystème riche.

7. Conclusion

Le cache est un outil essentiel pour améliorer les performances et la scalabilité des systèmes applicatifs modernes. Utilisé naïvement comme simple cache local, il apporte un gain de perfor-

mance mais pose rapidement des problèmes de cohérence, de duplication et d'invalidation dans les architectures multi-serveurs. Le cache distribué, et en particulier Redis, répond à ces enjeux en offrant un stockage en mémoire centralisé, partagé entre toutes les instances de l'application.

Les patterns de cache (cache-aside, read-through, write-through, write-behind) fournissent un vocabulaire et des solutions éprouvées pour structurer les interactions entre l'application, le cache et la base de données. Redis s'intègre naturellement dans ces patterns, notamment cache-aside, et s'impose aujourd'hui comme un composant clé dans de nombreuses architectures cloud et microservices, où la performance et la réactivité sont des priorités.

Références

— **Redis, "Caching" (documentation officielle)** : `https://redis.io/solutions/caching/`.
Utilisé principalement pour les sections sur la définition de la mise en cache et le rôle de Redis comme cache distribué (sections 1, 3, 4 et 6).

— **IBM, "What is Redis ?"** : `https://www.ibm.com/think/topics/redis`.
Source pour la présentation générale de Redis comme *in-memory* data store, ses cas d'usage (cache, base primaire) et ses caractéristiques (section 4).

— **Azure Architecture Center, "Modèle Cache-Aside"** : `https://learn.microsoft.com/fr-fr/azure/architecture/patterns/cache-aside`.
Référence utilisée pour décrire le pattern cache-aside et l'opposer au cache local, notamment dans la section sur les patterns de cache (section 5).

— **AWS, "Database Caching Strategies Using Redis – Caching patterns"** : `https://docs.aws.amazon.com/whitepapers/latest/database-caching-strategies-using-redis/caching-patterns.html`.
Utilisé pour préciser les différents patterns (cache-aside, read-through, write-through, write-behind) et les bonnes pratiques associées (section 5).

— **Redis, "Redis vs Memcached"** : `https://redis.io/compare/memcached/`.
Source de comparaison entre Redis et Memcached, citée pour justifier le choix de Redis comme cache distribué de référence (section 6).

— **Kinsta, "Memcached vs Redis : Choose Your In-Memory Cache"** : `https://kinsta.com/blog/memcached-vs-redis/`.
Comparatif complémentaire utilisé pour illustrer les différences de fonctionnalités et de cas d'usage entre Redis et Memcached (section 6).