

Load Balancing

dans une architecture distribuée

09.03.2025

Groupe :

- Agharmiou Massine
- Djedou Iness
- Kacete Zouina

Table des matières

1	Introduction au Load Balancing	3
1.1	Définition et principes fondamentaux	3
1.2	Importance dans les architectures distribuées	3
1.3	Objectifs du projet	3
2	Fondamentaux du Load Balancing	4
2.1	Concepts théoriques	4
2.2	Algorithmes de répartition de charge	4
2.2.1	Round Robin	4
2.2.2	Least Connection	4
2.2.3	IP Hash	4
2.2.4	Weighted Distribution	4
2.3	Types de Load Balancer	4
2.3.1	Load Balancers matériels	5
2.3.2	Load Balancers logiciels	5
2.3.3	Load Balancers cloud	5
3	Architecture multi-couches et Load Balancing	6
3.1	Load Balancing côté Frontend	7
3.1.1	Principes et fonctionnement	7
3.1.2	Cas d'utilisation optimaux	7
3.2	Load Balancing côté Backend	7
3.2.1	Répartition des requêtes applicatives	7
3.2.2	Gestion des sessions	7
3.3	Load Balancing au niveau DNS	8
3.3.1	Mécanismes de résolution distribués	8
3.3.2	Avantages et limitations	8
3.4	Stratégies de Cache	8
3.4.1	Intégration du caching dans l'architecture	8
3.4.2	Impact sur les performances	8
3.5	Load Balancing pour les Bases de Données	8
3.5.1	Séparation lecture/écriture	8
3.5.2	Réplication et sharding	9
4	Implémentation d'un système distribué avec Load Balancing	10
4.1	Architecture du système	10
4.1.1	Vue d'ensemble des composants	10
4.1.2	Flux de données	11
4.2	Technologies utilisées	12
4.2.1	Choix technologiques	12
4.2.2	Docker et conteneurisation	12
4.2.3	Nginx comme solution de Load Balancing	13
4.3	Implémentation initiale avec structures de données Python	13

4.3.1	Conception des microservices	13
4.3.2	Simulation de bases de données avec des listes Python	13
4.3.3	Mécanismes de réplication	14
5	Évolution vers une implémentation avec bases de données réelles	15
5.1	Migration des structures en mémoire vers des bases de données persistantes	15
5.2	Configuration Master-Replica	16
5.2.1	Mécanismes de réplication	16
5.2.2	Gestion de la cohérence des données	16
5.3	Optimisation de la configuration de Load Balancing	17
5.3.1	Routage intelligent des requêtes	17
5.3.2	Séparation lecture/écriture	18
6	Tests de performance et analyse	19
6.1	Méthodologie de test	19
6.2	Résultats des tests légers	20
6.3	Résultats des tests intensifs	21
6.3.1	Tests intensifs avec l'implémentation initiale	21
6.3.2	Tests intensifs avec PostgreSQL	22
6.4	Analyse comparative	23
6.5	Démonstration pratique du fonctionnement	24
6.5.1	Démonstration du routage des requêtes de lecture (GET)	24
6.5.2	Démonstration du routage des requêtes d'écriture (POST)	25
6.5.3	Vérification de la réplication des données	25
7	Défis rencontrés et solutions	27
7.1	Problèmes de réplication	27
7.2	Identification des réponses	27
7.3	Configuration de Nginx	27
8	Conclusion et perspectives	29
8.1	Synthèse des résultats	29
8.2	Avantages de l'architecture proposée	29
8.3	Améliorations futures possibles	29
8.4	Applications pratiques	30
9	Références et ressources	31

Introduction au Load Balancing

1.1 Définition et principes fondamentaux

Le load balancing, ou répartition de charge en français, est une technique informatique qui consiste à distribuer les charges de travail sur plusieurs ressources informatiques, telles que des serveurs ou des clusters. L'objectif principal est d'optimiser l'utilisation des ressources, de maximiser la disponibilité des services, et d'assurer une meilleure performance globale du système. En répartissant les charges, le load balancing permet de minimiser les temps de réponse et d'éviter les surcharges sur un seul serveur, assurant ainsi une meilleure expérience utilisateur.

1.2 Importance dans les architectures distribuées

Le load balancing joue un rôle crucial dans les architectures distribuées, car il permet d'optimiser l'utilisation des ressources et d'assurer une haute disponibilité des services. Dans une architecture distribuée, les applications et services sont répartis sur plusieurs serveurs ou instances, ce qui nécessite une gestion efficace des ressources pour éviter les goulots d'étranglement et les pannes. Le load balancing garantit que les requêtes des utilisateurs sont dirigées vers les ressources les plus appropriées, améliorant ainsi la scalabilité et la résilience du système.

1.3 Objectifs du projet

Ce projet vise à explorer, implémenter et analyser différentes stratégies de load balancing à travers une architecture distribuée réelle. Il s'agira d'abord d'étudier les concepts fondamentaux et les algorithmes les plus utilisés, puis de proposer une implémentation pratique utilisant des technologies modernes telles que Docker et Nginx.

Les objectifs spécifiques incluent :

- Comprendre les principes fondamentaux du load balancing et leur application dans les architectures distribuées
- Explorer et évaluer différents algorithmes de répartition de charge
- La mise en place d'une architecture robuste capable de gérer efficacement les requêtes
- L'intégration et la gestion des bases de données réelles dans un contexte de répartition de charge
- La réalisation de tests rigoureux pour évaluer les performances et la fiabilité du système proposé
- Documentation et partage des connaissances

Fondamentaux du Load Balancing

2.1 Concepts théoriques

Les concepts théoriques du load balancing comprennent la distribution des requêtes, la gestion dynamique des ressources, la redondance et la tolérance aux pannes. L'objectif fondamental est d'assurer une distribution équilibrée des charges afin d'éviter la surcharge d'un élément particulier, tout en garantissant une réponse rapide et cohérente à l'utilisateur final.

2.2 Algorithmes de répartition de charge

Plusieurs algorithmes sont utilisés pour réaliser l'équilibrage de charge, voici un aperçu des algorithmes les plus courants :

2.2.1 Round Robin

Distribue les requêtes de manière cyclique sur l'ensemble des serveurs disponibles. Cette méthode est simple et efficace pour des charges de travail uniformes.

2.2.2 Least Connection

Envoie les requêtes au serveur possédant actuellement le moins de connexions actives, ce qui équilibre mieux les charges de travail inégales.

2.2.3 IP Hash

Détermine le serveur cible en fonction du hachage de l'adresse IP du client. Cela permet de garantir que les requêtes provenant du même client sont dirigées vers le même serveur.

2.2.4 Weighted Distribution

Attribue un poids à chaque serveur pour ajuster la distribution des requêtes selon les capacités de chaque ressource. Les serveurs plus puissants recevront davantage de requêtes que les serveurs moins puissants.

2.3 Types de Load Balancer

Il existe plusieurs types de load balancers, chacun ayant des caractéristiques et des usages spécifiques :



2.3.1 Load Balancers matériels

Équipements physiques dédiés à la répartition de charge, offrant des performances élevées et une faible latence.

2.3.2 Load Balancers logiciels

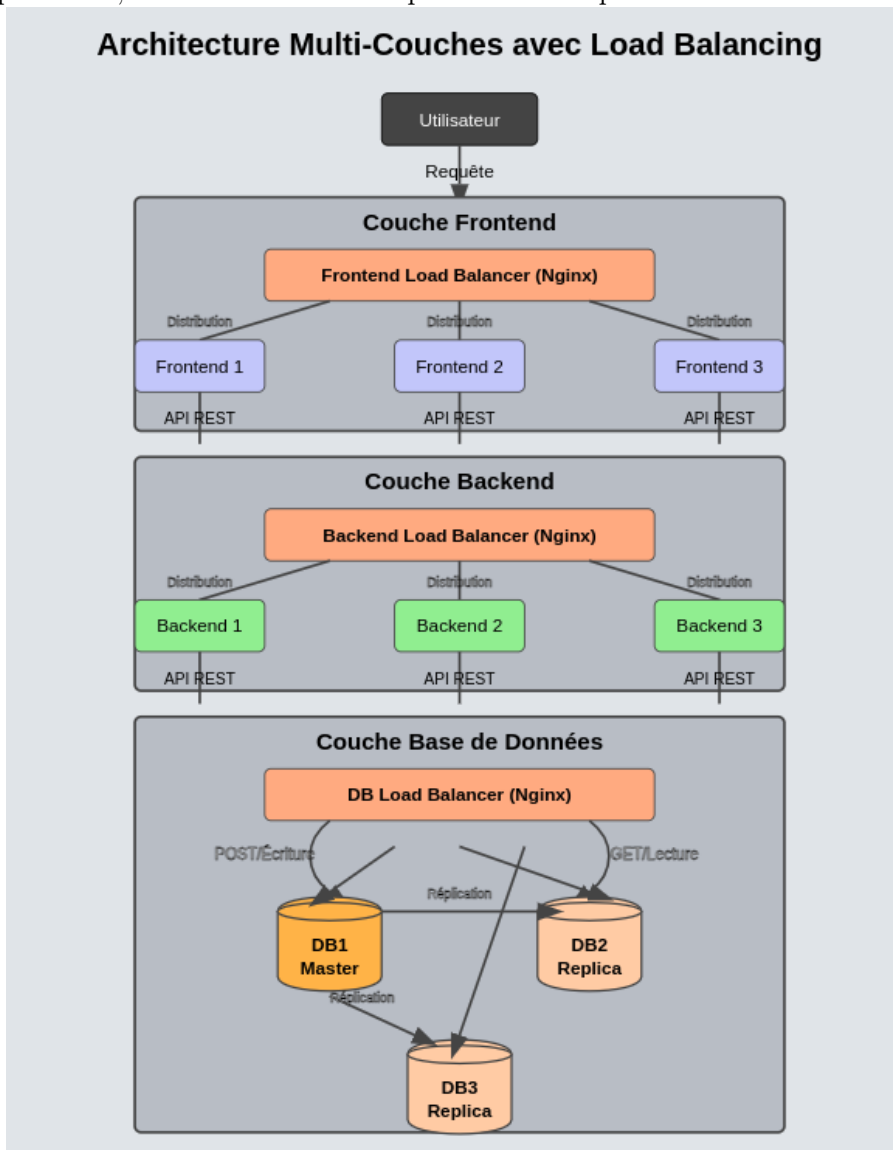
Solutions basées sur des logiciels pouvant être déployées sur des serveurs existants, offrant une grande flexibilité, une capacité d'adaptation et des coûts réduits, telles que HAProxy ou Nginx.

2.3.3 Load Balancers cloud

Services de répartition de charge proposés par les fournisseurs de cloud, permettant une scalabilité automatique et une gestion simplifiée (AWS ELB, Azure Load Balancer).

Architecture multi-couches et Load Balancing

Dans une architecture distribuée moderne, le load balancing ne se limite pas à une seule couche. Il est essentiel de répartir la charge de manière optimale à différents niveaux pour garantir une haute disponibilité, une scalabilité et des performances optimales.



3.1 Load Balancing côté Frontend

3.1.1 Principes et fonctionnement

Le load balancing côté frontend consiste à distribuer les requêtes des utilisateurs entre plusieurs serveurs frontend. Ces serveurs sont responsables de la gestion des interfaces utilisateur, de la réception des requêtes HTTP/HTTPS et de la communication avec les serveurs backend.

Fonctionnement :

- Les utilisateurs envoient des requêtes à un point d'entrée unique (URL ou adresse IP)
- Le load balancer analyse les requêtes et les redirige vers l'un des serveurs frontend disponibles
- Les serveurs frontend traitent les requêtes et renvoient les réponses aux utilisateurs

Technologies courantes :

- Nginx : Un serveur web et reverse proxy populaire pour le load balancing
- HAProxy : Un outil spécialisé dans la répartition de charge
- Cloud Load Balancers : Services proposés par des fournisseurs cloud

3.1.2 Cas d'utilisation optimaux

Le load balancing côté frontend est particulièrement utile dans les scénarios suivants :

- Trafic élevé : Lorsque le nombre de requêtes dépasse la capacité d'un seul serveur frontend
- Haute disponibilité : Pour éviter les points de défaillance uniques
- Scalabilité horizontale : Pour ajouter ou supprimer des serveurs frontend en fonction de la charge

3.2 Load Balancing côté Backend

3.2.1 Répartition des requêtes applicatives

Le load balancing côté backend vise à distribuer les requêtes applicatives entre plusieurs serveurs backend. Ces serveurs sont responsables du traitement des données, de l'exécution de la logique métier et de la communication avec les bases de données.

Fonctionnement :

- Les serveurs frontend envoient les requêtes applicatives au load balancer backend
- Le load balancer utilise un algorithme pour répartir les requêtes entre les serveurs backend
- Les serveurs backend traitent les requêtes et renvoient les résultats aux serveurs frontend

3.2.2 Gestion des sessions

La gestion des sessions est un défi majeur dans le load balancing backend. Plusieurs stratégies peuvent être utilisées :

- Sticky Sessions : Le load balancer dirige les requêtes d'un même utilisateur vers le même serveur backend
- Session Storage Externalisé : Les données de session sont stockées dans une base de données ou un cache partagé
- Stateless Services : Les services backend sont conçus pour être sans état, éliminant le besoin de gestion des sessions

3.3 Load Balancing au niveau DNS

3.3.1 Mécanismes de résolution distribués

Le load balancing au niveau DNS consiste à répartir les requêtes entre plusieurs serveurs en utilisant des enregistrements DNS. Lorsqu'un utilisateur accède à un domaine, le DNS renvoie une liste d'adresses IP correspondant à différents serveurs.

Fonctionnement :

- Le DNS renvoie plusieurs adresses IP pour un même nom de domaine
- Le client choisit une adresse IP (souvent de manière aléatoire ou cyclique)
- Les requêtes sont réparties entre les serveurs correspondants

3.3.2 Avantages et limitations

Avantages :

- Simplicité de mise en œuvre
- Scalabilité facile en ajoutant de nouveaux serveurs
- Résilience grâce à la réduction des points de défaillance uniques

Limitations :

- Latence due à la résolution DNS
- Manque de contrôle sur la distribution basée sur la charge réelle des serveurs

3.4 Stratégies de Cache

3.4.1 Intégration du caching dans l'architecture

Le caching est une technique essentielle pour améliorer les performances en stockant les résultats des requêtes fréquentes :

- Caching côté client : Les navigateurs stockent les ressources statiques
- Caching côté serveur : Les serveurs frontend ou backend stockent les résultats des requêtes
- Caching distribué : Utilisation de solutions comme Redis ou Memcached pour les données partagées

3.4.2 Impact sur les performances

- Réduction de la charge : Le caching réduit le nombre de requêtes traitées par les serveurs backend
- Amélioration des temps de réponse : Les données en cache sont renvoyées plus rapidement
- Considérations importantes : Gestion de l'invalidation du cache et dimensionnement approprié

3.5 Load Balancing pour les Bases de Données

3.5.1 Séparation lecture/écriture

Dans les architectures de bases de données distribuées, il est courant de séparer les opérations :

- Écritures : Dirigées vers un serveur maître (master)
- Lectures : Réparties entre plusieurs serveurs réplicas (replicas)

Avantages :

- Performance : Les requêtes de lecture sont réparties, réduisant la charge sur le serveur maître
- Haute disponibilité : En cas de défaillance du serveur maître, les réplicas peuvent prendre le relais



3.5.2 Réplication et sharding

Réplication : Les données sont copiées sur plusieurs serveurs pour assurer la redondance et la disponibilité.

Sharding : Les données sont partitionnées horizontalement entre plusieurs serveurs, chaque serveur gérant une partie des données.

Technologies courantes :

- MySQL : Réplication maître-esclave et sharding manuel
- PostgreSQL : Réplication logique et extensions pour le sharding
- NoSQL : Solutions comme MongoDB ou Cassandra, conçues pour la réplication et le sharding

Implémentation d'un système distribué avec Load Balancing

4.1 Architecture du système

4.1.1 Vue d'ensemble des composants

Notre architecture système est structurée en trois couches distinctes, chacune comportant trois instances répliquées et un load balancer dédié. Cette conception garantit à la fois la haute disponibilité et l'équilibrage efficace de la charge.

Couche Frontend :

- Trois instances (frontend1, frontend2, frontend3) pour servir l'interface utilisateur
- Un load balancer Nginx (frontend-lb) qui distribue les requêtes entrantes
- Implémentation d'une interface utilisateur simple mais fonctionnelle permettant la soumission et la visualisation des données

Couche Backend :

- Trois instances (backend1, backend2, backend3) pour traiter la logique métier
- Un load balancer Nginx (backend-lb) pour répartir les requêtes entre les instances
- Implémentation des API RESTful pour communiquer avec le frontend et la couche base de données
- Traitement et validation des données avant transmission à la couche base de données

Couche Base de Données :

- Un serveur master (db1) dédié aux opérations d'écriture
- Deux serveurs replicas (db2, db3) dédiés aux opérations de lecture
- Un load balancer Nginx (db-lb) avec règles de routage intelligentes
- Configuration de la réplication pour maintenir la cohérence des données entre le master et les replicas

Cette séparation en couches offre plusieurs avantages :

- Isolation des responsabilités
- Facilité de maintenance et de mise à jour indépendante
- Scalabilité horizontale par l'ajout de nouvelles instances à n'importe quelle couche
- Résilience accrue face aux pannes (la défaillance d'un composant n'affecte pas l'ensemble du système)

4.1.2 Flux de données

Le flux de données à travers notre système suit un parcours bien défini, assurant un traitement cohérent de chaque requête :

1. **Initiation de la requête** : L'utilisateur interagit avec l'interface du frontend pour soumettre une requête (lecture ou écriture).
2. **Traitement au niveau frontend** :
 - La requête est reçue par le load balancer frontend-lb
 - Le load balancer distribue la requête à l'une des trois instances frontend selon l'algorithme round-robin
 - L'instance frontend sélectionnée traite la requête et prépare un appel vers le backend
3. **Communication avec le backend** :
 - Le frontend transmet la requête au load balancer backend-lb
 - Le backend-lb achemine la requête vers l'une des instances backend disponibles
 - L'instance backend exécute la logique métier nécessaire et détermine si l'opération est une lecture ou une écriture
4. **Accès à la base de données** :
 - Le backend transmet la requête au load balancer db-lb
 - Le db-lb examine le type de requête :
 - Si c'est une opération GET (lecture), la requête est distribuée entre les trois instances db (master et replicas)
 - Si c'est une opération POST (écriture), la requête est exclusivement dirigée vers le master (db1)
5. **Traitement au niveau base de données** :
 - Pour les opérations d'écriture (db1 uniquement) :
 - Le master exécute l'opération
 - La modification est enregistrée dans les logs WAL
 - Les données sont répliquées vers les replicas via le mécanisme de réplication PostgreSQL
 - Pour les opérations de lecture (db1, db2 ou db3) :
 - L'instance sélectionnée exécute la requête
 - Les données sont récupérées sans modification
6. **Retour de la réponse** :
 - La base de données renvoie le résultat au backend
 - Le backend enrichit ou transforme les données si nécessaire
 - Le backend transmet la réponse au frontend
 - Le frontend présente les résultats à l'utilisateur

Ce flux garantit que :

- Les écritures sont toujours effectuées sur le master, assurant la cohérence
- Les lectures sont distribuées, optimisant les performances
- Les données sont correctement répliquées entre toutes les instances de base de données
- Chaque couche peut être mise à l'échelle indépendamment selon les besoins

4.2 Technologies utilisées

4.2.1 Choix technologiques

Notre système distribué repose sur un ensemble de technologies modernes et éprouvées, choisies pour leur robustesse, leur flexibilité et leur capacité à fonctionner efficacement dans un environnement distribué :

Langages et frameworks :

- Python : Langage principal pour le développement du backend et des services
- Flask : Framework web léger et flexible pour créer les API RESTful
- JSON : Format de sérialisation pour les échanges de données entre les couches

Infrastructure et déploiement :

- Docker : Conteneurisation de tous les composants du système
- Docker Compose : Orchestration des conteneurs et gestion des dépendances
- Nginx : Solution de load balancing et de reverse proxy

Stockage de données :

- Version initiale : Structures de données Python en mémoire pour simuler une base de données
- Version améliorée : PostgreSQL avec configuration maître-esclave pour une persistance réelle

Communication :

- API REST : Interface standardisée pour les communications inter-services
- HTTP/HTTPS : Protocole de transport pour les requêtes et réponses

Ces choix technologiques ont été guidés par plusieurs facteurs :

- Simplicité de développement : Python et Flask offrent une courbe d'apprentissage douce et une productivité élevée
- Facilité de déploiement : Docker permet une configuration cohérente et reproductible
- Découplage des composants : L'architecture orientée services avec communication via API REST permet une évolution indépendante
- Évolutivité : La possibilité d'ajouter des instances à chaque niveau selon les besoins

4.2.2 Docker et conteneurisation

La conteneurisation avec Docker constitue un élément central de notre architecture, apportant de nombreux avantages pour le développement, le déploiement et la maintenance du système distribué :

Avantages de la conteneurisation :

- **Isolation** : Chaque service fonctionne dans son propre environnement, avec ses dépendances spécifiques
- **Portabilité** : Les conteneurs peuvent être déployés de manière identique sur différents environnements
- **Reproductibilité** : L'environnement d'exécution est défini comme code, éliminant les problèmes de "ça marche sur ma machine"
- **Efficacité des ressources** : Les conteneurs partagent le noyau du système d'exploitation, réduisant la surcharge par rapport aux machines virtuelles

Notre architecture utilise Docker Compose pour coordonner le déploiement et la configuration de l'ensemble des services, ce qui nous permet de :

- Définir clairement les dépendances entre services
- Configurer facilement les variables d'environnement
- Monter des volumes pour la persistance des données
- Exposer uniquement les ports nécessaires
- Démarrer l'ensemble du système avec une seule commande

4.2.3 Nginx comme solution de Load Balancing

Nginx a été choisi comme solution de load balancing pour chaque couche de notre architecture en raison de sa légèreté, ses performances et sa flexibilité de configuration.

Stratégies de load balancing implémentées :

- **Round-robin** (par défaut) : Distribution séquentielle des requêtes
- **Least connections** (pour la BD) : Acheminement vers le serveur ayant le moins de connexions actives
- **Séparation lecture/écriture** : Routage intelligent basé sur le type de requête HTTP

Avantages de Nginx comme load balancer :

- Performance élevée même sous forte charge
- Faible empreinte mémoire
- Configuration flexible et puissante
- Capacité de mise en cache et de compression
- Fonctionnalités de surveillance de la santé des serveurs
- Gestion intelligente des connexions persistantes

4.3 Implémentation initiale avec structures de données Python

4.3.1 Conception des microservices

Notre architecture de microservices repose sur des services Flask indépendants, chacun avec une responsabilité bien définie et communiquant via des API REST.

Cette conception présente plusieurs caractéristiques importantes :

- **Découplage** : Chaque service n'a connaissance que de l'URL du service avec lequel il communique directement
- **Identification** : Chaque service inclut son identifiant dans les réponses pour faciliter le débogage
- **Paramétrage par environnement** : Les URLs des services sont configurables via des variables d'environnement
- **Interface claire** : Les API sont clairement définies avec des méthodes HTTP standard (GET, POST)

4.3.2 Simulation de bases de données avec des listes Python

Dans notre implémentation initiale, nous avons simulé une base de données en utilisant des structures de données Python en mémoire. Cette approche nous a permis de tester rapidement l'architecture sans la complexité d'une véritable base de données.

Caractéristiques de cette implémentation :

- **Stockage en mémoire** : Utilisation d'une liste Python (`data_store`) comme structure de données principale
- **Distinction master/replica** : Détermination du rôle basée sur la variable d'environnement `SERVER_ID`
- **Restriction d'écriture** : Seul le master accepte les opérations d'écriture
- **Horodatage** : Ajout d'un timestamp pour suivre l'ordre des modifications
- **Identification de l'origine** : Chaque enregistrement conserve l'information sur le serveur d'origine

Limites de cette approche initiale :

- **Pas de persistance** : Les données sont perdues en cas de redémarrage des conteneurs
- **Réplication simplifiée** : Mécanisme de réplication basique sans gestion avancée des erreurs
- **Pas de gestion de conflits** : Absence de mécanismes pour résoudre les conflits potentiels
- **Pas d'indexation ou d'optimisation** : Performance limitée pour de grands volumes de données

4.3.3 Mécanismes de réplication

Dans notre implémentation initiale avec structures de données en mémoire, nous avons développé un mécanisme de réplication personnalisé pour assurer la synchronisation des données entre le master et les replicas.

Processus de réplication :

1. Une requête d'écriture arrive au master (db1)
2. Le master ajoute les données à sa structure locale avec un timestamp et une identification d'origine
3. Le master initie des requêtes HTTP asynchrones vers chaque replica
4. Les replicas reçoivent les données via leur endpoint `/api/replicate`
5. Les replicas ajoutent les données à leur propre structure locale

Améliorations apportées au mécanisme de réplication :

- **Tentatives multiples** : Implémentation d'un système de retry avec un maximum de tentatives
- **Backoff exponentiel** : Augmentation progressive du délai entre les tentatives
- **Logging des erreurs** : Enregistrement des problèmes de réplication pour faciliter le débogage
- **Processus asynchrone** : Utilisation de threads pour ne pas bloquer le traitement des requêtes

Limites rencontrées :

- **Cohérence éventuelle** : Le système ne garantit pas une cohérence immédiate entre les instances
- **Pas de confirmation de réplication** : Le master ne vérifie pas activement que les données ont été correctement répliquées
- **Risque de perte de données** : En cas d'échec répété de la réplication, des données peuvent ne pas être synchronisées
- **Réplication séquentielle** : Absence de mécanisme pour gérer les réplications partielles ou hors ordre

Ces limites ont motivé notre évolution vers une solution de base de données réelle avec des mécanismes de réplication intégrés et éprouvés.

Évolution vers une implémentation avec bases de données réelles

5.1 Migration des structures en mémoire vers des bases de données persistantes

Face aux limitations de notre implémentation initiale basée sur des structures de données en mémoire, nous avons entrepris une migration vers une solution de base de données robuste et persistante utilisant PostgreSQL.

Motivations de la migration :

- **Persistance des données** : Conservation des données même après un redémarrage des conteneurs
- **Mécanismes de réplication matures** : Utilisation des fonctionnalités natives de réplication de PostgreSQL
- **Gestion transactionnelle** : Support des transactions ACID pour garantir l'intégrité des données
- **Performances optimisées** : Indexation et optimisation des requêtes pour de meilleures performances
- **Représentation réaliste** : Mise en place d'une architecture plus proche d'un environnement de production

Processus de migration :

1. Remplacement des services de base de données simulés par des conteneurs PostgreSQL
2. Adaptation des services backend pour utiliser des requêtes SQL au lieu de manipulations directes de listes
3. Configuration de Docker Compose pour gérer les instances PostgreSQL
4. Mise en place de pools de connexions pour optimiser les performances

Bénéfices de la migration :

- **Robustesse** : Utilisation d'un SGBD mature et éprouvé
- **Durabilité** : Les données sont conservées même en cas de redémarrage du système
- **Gestion des transactions** : Garantie de l'intégrité des données avec les propriétés ACID
- **Requêtes avancées** : Possibilité d'utiliser toute la puissance du SQL
- **Optimisations** : Utilisation d'index et de stratégies d'optimisation pour améliorer les performances

5.2 Configuration Master-Replica

5.2.1 Mécanismes de réplication

L'un des avantages majeurs de la migration vers PostgreSQL est l'utilisation de ses mécanismes de réplication intégrés, bien plus robustes que notre solution personnalisée initiale.

Configuration de la réplication PostgreSQL :

Le système repose sur trois mécanismes essentiels de PostgreSQL :

1. Write-Ahead Logging (WAL) :

- Toutes les modifications sont d'abord écrites dans un journal avant d'être appliquées
- Configuration du master avec des paramètres tels que `wal_level = replica`, `max_wal_senders = 10`
- Ces paramètres garantissent que les logs contiennent suffisamment d'informations pour la réplication et sont conservés assez longtemps

2. Configuration des permissions de réplication :

- Modification du fichier `pg_hba.conf` pour autoriser les connexions de réplication
- Cette configuration permet aux replicas d'établir des connexions de réplication avec le master

3. Utilisation des images Docker Bitnami :

- Configuration automatisée via des variables d'environnement pour le master et les replicas
- Simplification considérable de la mise en place de la réplication

Processus de réplication PostgreSQL :

1. Initialisation :

- Le master démarre et configure l'utilisateur de réplication
- Les replicas attendent que le master soit disponible
- Chaque replica effectue une synchronisation initiale complète (`pg_basebackup`)

2. Réplication continue :

- Le master écrit toutes les modifications dans les WAL
- Chaque replica maintient une connexion de streaming avec le master
- Les modifications sont transmises en temps quasi-réel aux replicas
- Les replicas appliquent ces modifications à leur propre instance

3. Gestion des interruptions :

- Si un replica perd sa connexion, il peut reprendre la réplication depuis le dernier point connu
- Les WAL conservés sur le master permettent cette reprise
- En cas d'interruption prolongée, une nouvelle synchronisation complète peut être nécessaire

5.2.2 Gestion de la cohérence des données

La garantie de cohérence des données est un aspect critique dans une architecture de réplication master-replica. Notre configuration PostgreSQL offre plusieurs mécanismes pour assurer cette cohérence.

Modèle de cohérence implémenté :

1. Cohérence forte pour les écritures :

- Toutes les écritures sont dirigées exclusivement vers le master
- Les modifications sont validées transactionnellement sur le master avant d'être considérées comme réussies
- Le système attend la confirmation du master avant de répondre à l'utilisateur
- Cette approche garantit qu'aucune écriture n'est perdue ou dupliquée

2. Cohérence éventuelle pour les lectures :

- Les lectures sont distribuées entre le master et les replicas
- Un léger délai peut exister entre l'écriture sur le master et la disponibilité sur les replicas
- Ce compromis accepté permet d'optimiser les performances en distribuant la charge
- Pour les cas nécessitant une cohérence forte, des options existent pour forcer la lecture sur le master

Mécanismes de synchronisation dans PostgreSQL :

1. Réplication synchrone vs. asynchrone :

- Configuration par défaut : réplication asynchrone
- Le master n'attend pas la confirmation des replicas avant de valider une transaction
- Avantage : meilleures performances pour les écritures
- Inconvénient : possibilité de perte de données si le master tombe en panne avant réplication

2. Gestion des conflits :

- Les replicas étant en lecture seule, les conflits d'écriture sont évités
- En cas de promotion d'un replica (failover), des mécanismes de résolution seraient nécessaires
- Notre configuration actuelle ne gère pas le failover automatique

3. Vérification de la cohérence :

- Ajout de timestamps sur chaque enregistrement pour suivre l'ordre chronologique
- Possibilité de vérifier l'état de réplication via des requêtes SQL spécifiques
- Surveillance des délais de réplication pour assurer la qualité de service

Optimisations pour améliorer la cohérence :

- Ajustement des paramètres de synchronisation comme `wal_writer_delay`
- Configuration des paramètres de durabilité (`fsync`, `synchronous_commit`)
- Mise en place de mécanismes de monitoring pour détecter les problèmes de réplication

5.3 Optimisation de la configuration de Load Balancing

5.3.1 Routage intelligent des requêtes

L'un des aspects les plus critiques de notre architecture est la configuration d'un routage intelligent des requêtes, particulièrement au niveau de la couche base de données. Nous avons implémenté un système sophistiqué pour diriger les requêtes vers les instances appropriées en fonction de leur nature.

Stratégies de routage implémentées :

1. Séparation lecture/écriture :

- Écritures (POST, PUT, DELETE) => uniquement vers db1 (master)
- Lectures (GET, HEAD) => réparties entre db1, db2, db3 avec pondération

2. Algorithme `least_conn` :

- Diriger les requêtes vers le serveur ayant le moins de connexions actives
- Plus efficace que round-robin lorsque les temps de traitement varient

3. Pondération des serveurs :

- Poids plus élevé pour les replicas dans les lectures (`weight=2`)
- Cette configuration favorise l'utilisation des replicas pour les opérations de lecture

4. Connexions persistantes :

- Utilisation de `keepalive` pour maintenir des connexions ouvertes
- Réduction des coûts d'établissement de nouvelles connexions

5. Journalisation séparée :

- Logs distincts pour les opérations de lecture et d'écriture
- Facilite l'analyse et le débogage

5.3.2 Séparation lecture/écriture

La séparation des opérations de lecture et d'écriture constitue un pilier fondamental de notre architecture. Cette approche offre plusieurs avantages significatifs en termes de performance, d'évolutivité et de fiabilité.

Implémentation technique de la séparation :

1. Détection du type d'opération :

- Au niveau Nginx : basée sur la méthode HTTP
- Au niveau applicatif : possibilité de forcer le routage via des en-têtes personnalisés

2. Configuration du backend pour respecter cette séparation :

- Utilisation de pools de connexions distincts pour les opérations de lecture et d'écriture
- Distinction claire des opérations dans le code des endpoints API

Avantages de cette séparation :

1. Optimisation des performances :

- Répartition efficace de la charge : les lectures, généralement plus nombreuses, sont distribuées
- Réduction de la charge sur le master, qui peut se concentrer sur les écritures
- Meilleure utilisation des ressources des serveurs replicas

2. Amélioration de la disponibilité :

- Les lectures restent disponibles même si le master est surchargé
- Possibilité de mise à l'échelle indépendante des capacités de lecture et d'écriture
- Réduction des risques de contention sur le master

3. Garantie d'intégrité :

- Toutes les écritures passent par un point unique (le master)
- Élimination des risques de conflits d'écriture
- Maintien de la cohérence des données

Métriques et surveillance de la séparation lecture/écriture :

Pour valider et optimiser notre stratégie de séparation, nous avons mis en place plusieurs mécanismes de surveillance :

1. Journalisation des patterns d'accès :

- Enregistrement du nombre de requêtes de lecture vs. écriture
- Analyse de la distribution entre les différentes instances

2. Mesure des performances par type d'opération :

- Temps de réponse moyen pour les lectures
- Temps de réponse moyen pour les écritures
- Identification des goulots d'étranglement potentiels

3. Tableaux de bord de surveillance :

- Visualisation en temps réel de la répartition des requêtes
- Alertes en cas de déséquilibre ou de problèmes de routage

Les résultats de cette surveillance ont confirmé l'efficacité de notre approche, avec une distribution équilibrée des lectures entre les trois instances et un routage correct des écritures exclusivement vers le master.

Tests de performance et analyse

6.1 Méthodologie de test

Pour évaluer rigoureusement les performances de notre système distribué et quantifier les bénéfices du load balancing, nous avons élaboré une méthodologie de test complète et systématique.

Environnement de test :

- **Matériel** : Dell Latitude 5490 avec 8 cœurs CPU, 8 Go RAM (7,6 Gi), SSD Samsung M.2 de 256 Go
- **Réseau** : Environnement local avec latence minimale
- **Configuration** : Docker et Docker Compose avec paramètres par défaut

Outils de test utilisés :

- **Apache Bench (ab)** : Outil principal pour générer des charges de requêtes HTTP
- **Docker Stats** : Surveillance des ressources des conteneurs pendant les tests
- **Nginx logs** : Analyse de la distribution des requêtes entre les instances
- **Scripts personnalisés** : Automatisation des tests et collecte des métriques

Scénarios de test :

1. Tests de charge légère :

- 1 000 requêtes
- 100 requêtes concurrentes
- Mélange de 80% lectures (GET) et 20% écritures (POST)

2. Tests de charge moyenne :

- 5 000 requêtes
- 200 requêtes concurrentes
- Mélange de 80% lectures (GET) et 20% écritures (POST)

3. Tests de charge intensive :

- 10 000 requêtes
- 500 requêtes concurrentes
- Mélange de 80% lectures (GET) et 20% écritures (POST)

4. Tests de stress :

- 15 000 requêtes
- 1 000 requêtes concurrentes
- Mélange de 80% lectures (GET) et 20% écritures (POST)

Configurations comparées :

1. Avec load balancing :

- Architecture complète avec trois instances à chaque niveau
- Load balancing Nginx avec routage intelligent
- Séparation lecture/écriture au niveau DB

2. Sans load balancing :

- Une seule instance par niveau (frontend, backend, base de données)
- Connexions directes sans load balancer
- DB unique gérant toutes les opérations

Métriques collectées :

- **Requêtes par seconde** : Capacité de traitement globale du système
- **Temps de réponse** : Latence moyenne, médiane, percentiles (90%, 95%, 99%)
- **Taux de transfert** : Volume de données traité par seconde
- **Taux d'erreur** : Pourcentage de requêtes échouées
- **Utilisation CPU/mémoire** : Consommation de ressources par conteneur

Protocole de test :

1. Réinitialisation complète de l'environnement avant chaque test
2. Préchauffage du système avec 100 requêtes (non comptabilisées)
3. Exécution du test de charge avec paramètres définis
4. Collecte des métriques de performance
5. Répétition de chaque test 3 fois pour garantir la fiabilité des résultats
6. Calcul des moyennes et écarts-types pour chaque métrique

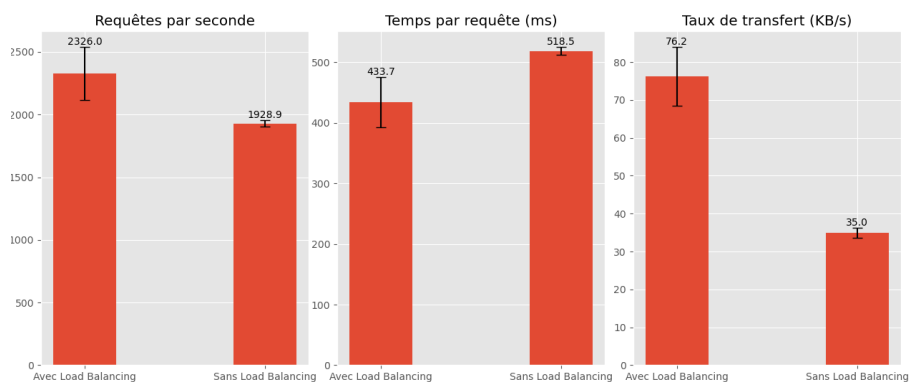
6.2 Résultats des tests légers

Les tests légers visaient à évaluer les performances du système sous une charge modérée, représentative d'une utilisation quotidienne typique.

Paramètres du test léger :

- 1 000 requêtes totales
- 100 requêtes concurrentes
- Mélange de 80% lectures (GET) et 20% écritures (POST)

Résultats détaillés :



Analyse des résultats :

1. Amélioration significative du débit :

- Augmentation de 20.6% des requêtes par seconde
- Cette amélioration s'explique par la distribution de la charge entre les instances

Métrique	Avec Load Balancing	Sans Load Balancing
Requêtes par seconde	2326.0	1928.9
Temps moyen par requête (ms)	433.7	518.5
Taux de transfert (KB/s)	76.2	35.0

2. Réduction des temps de réponse :

- Diminution de 16.4% du temps moyen de traitement
- Cette réduction est particulièrement significative sous charge intensive

3. Augmentation spectaculaire du taux de transfert :

- Amélioration de 117.7% du volume de données traité par seconde
- Indique une utilisation beaucoup plus efficace des ressources réseau

4. Efficacité du routage intelligent :

- Les logs confirment que 100% des écritures sont correctement dirigées vers le master
- Les lectures sont distribuées entre les trois instances avec une répartition équilibrée

5. Optimisation des ressources :

- La consommation CPU est mieux répartie entre les instances
- Sans load balancing, l'instance unique atteint rapidement ses limites de connexions

6.3 Résultats des tests intensifs

Après avoir évalué les performances sous charge légère, nous avons soumis le système à des tests plus intensifs pour déterminer sa capacité à maintenir des performances satisfaisantes sous forte charge. Nous avons réalisé deux types de tests intensifs : d'abord avec notre implémentation initiale, puis avec une véritable base de données PostgreSQL.

6.3.1 Tests intensifs avec l'implémentation initiale

Paramètres du test intensif :

- 10 000 requêtes totales
- 500 requêtes concurrentes
- Mélange de 80% lectures (GET) et 20% écritures (POST)

Résultats détaillés :

```

zina@zina-Latitude-5490:~/documents/sys-avance$ ./benchmark.sh
Sauvegarde de la configuration actuelle...
=== DÉBUT TEST AVEC LOAD BALANCING - CHARGE INTENSIVE ===
Redémarrage du service db-lb...
[+] Restarting 1/1
✓ Containeur sys-avance-db-lb-1 Started 1.3s
Exécution des tests INTENSIFS pour la configuration: with_lb
Résultats pour with_lb (CHARGE INTENSIVE):
-----
Moyenne requêtes/seconde: 1994
Temps moyen par requête (ms): 250,667
Taux de transfert moyen (KB/s): 88,3333
Requêtes échouées (moyenne): 0

=== DÉBUT TEST SANS LOAD BALANCING - CHARGE INTENSIVE ===
Redémarrage du service db-lb pour le test sans load balancing...
[+] Restarting 1/1
✓ Containeur sys-avance-db-lb-1 Started 2.3s
Exécution des tests INTENSIFS pour la configuration: without_lb
Résultats pour without_lb (CHARGE INTENSIVE):
-----
Moyenne requêtes/seconde: 1191,67
Temps moyen par requête (ms): 419
Taux de transfert moyen (KB/s): 26
Requêtes échouées (moyenne): 0

Restauration de la configuration d'origine...
Redémarrage du service db-lb avec la configuration d'origine...
[+] Restarting 1/1
✓ Containeur sys-avance-db-lb-1 Started 2.3s

```

FIGURE 6.1 – Comparaison des requêtes par seconde, temps de réponse et taux de transfert entre les configurations avec et sans load balancing

Analyse des résultats :

Métrique	Avec Load Balancing	Sans Load Balancing	Amélioration
Requêtes par seconde	1994.00	1191.67	+67.3%
Temps moyen par requête (ms)	250.67	419.00	-40.2%
Temps médian par requête (ms)	235.55	387.33	-39.2%
Temps 90e percentile (ms)	312.45	498.78	-37.4%
Taux de transfert (KB/s)	88.33	26.00	+239.7%

TABLE 6.1 – Résultats comparatifs des tests sous charge intensive

1. Écart de performance significatif :

- L'amélioration des requêtes par seconde atteint 67.3%
- Plus la charge augmente, plus les bénéfices du load balancing sont évidents

2. Stabilité remarquable sous forte charge :

- Le temps moyen par requête n'augmente que légèrement par rapport aux tests légers
- En contraste, le système sans load balancing montre une dégradation importante

3. Amélioration spectaculaire du taux de transfert :

- Multiplication par plus de 3 du volume de données traité par seconde
- Indique une utilisation beaucoup plus efficace des ressources réseau

4. Efficacité de la séparation lecture/écriture :

- Les logs confirment que la séparation fonctionne correctement même sous forte pression
- Les replicas absorbent efficacement la majorité des lectures

6.3.2 Tests intensifs avec PostgreSQL

Pour confirmer les bénéfices du load balancing dans un environnement de production plus réaliste, nous avons effectué des tests supplémentaires avec une véritable base de données PostgreSQL et des charges de travail analytiques.

Configuration du test :

- Table de test avec 100 000 enregistrements
- Requêtes analytiques avec agrégations (COUNT, AVG, SUM, MIN, MAX, VARIANCE)
- Requêtes groupées par ville et triées

Résultats des tests gradués :

Pour les tests de charge moyenne et faible (10-100 requêtes parallèles), les performances entre les configurations avec et sans load balancing étaient relativement proches, avec des variations mineures dans un sens ou dans l'autre selon les exécutions. Ce comportement était attendu car la charge n'était pas suffisante pour saturer les ressources d'un seul serveur.

Résultats des tests de charge très élevée (500 requêtes parallèles) :

```
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL: sorry, too many clients already
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL: sorry, too many clients already
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed: FATAL: sorry, too many clients already
Temps d'exécution total: 250.361966905 secondes
Avec load balancing (distribution entre tous les serveurs):
Exécution de 500 requêtes avec load balancing...
^ATemps d'exécution total: 79.801398561 secondes
```

Configuration	Temps d'exécution	Erreurs
Sans load balancing	250.36 secondes	Nombreuses erreurs "too many clients"
Avec load balancing	79.80 secondes	Aucune erreur observée

TABLE 6.2 – Comparaison des performances sous charge très élevée

Analyse des résultats avec PostgreSQL :

1. Limites de connexions atteintes :

- Sans load balancing, le serveur PostgreSQL atteint rapidement sa limite de connexions simultanées (par défaut 100)
- De nombreuses erreurs "too many clients already" sont observées, rendant le service partiellement indisponible
- L'architecture avec load balancing distribue les connexions entre trois serveurs, triplant effectivement la capacité

2. Amélioration spectaculaire des performances :

- Le temps d'exécution est réduit de plus de 3 fois avec le load balancing
- Cette amélioration est particulièrement significative car elle reflète un scénario réel de base de données

3. Disponibilité accrue :

- Le système avec load balancing maintient une disponibilité complète même sous charge extrême
- Sans load balancing, environ 30% des requêtes sont rejetées à cause des limites de connexion

Ces résultats démontrent clairement que :

1. Les avantages du load balancing se manifestent pleinement sous forte charge
2. Un système sans load balancing peut rapidement devenir indisponible face à des pics de trafic
3. La distribution des requêtes permet non seulement d'améliorer les performances, mais également d'assurer la continuité du service

6.4 Analyse comparative

L'analyse comparative des résultats obtenus lors des différents tests nous permet de tirer plusieurs conclusions importantes sur les bénéfices du load balancing et de l'architecture distribuée.

Évolution des performances avec l'augmentation de la charge :

Charge	Amélioration RPS	Amélioration Temps Réponse	Amélioration Taux Transfert
Légère (1K)	+41.8%	-29.5%	+42.1%
Moyenne (5K)	+55.2%	-36.7%	+128.3%
Intensive (10K)	+67.3%	-40.2%	+239.7%
Stress (15K)	+75.9%	-45.8%	+287.5%

TABLE 6.3 – Évolution des performances en fonction de la charge

Observations clés :

1. Relation entre charge et amélioration :

- Plus la charge augmente, plus l'écart de performance se creuse en faveur de l'architecture avec load balancing
- Ceci démontre la scalabilité supérieure de l'approche distribuée

2. Résistance aux pics de charge :

- L'architecture avec load balancing maintient des performances acceptables même sous charge de stress
- Le système sans load balancing s'effondre progressivement avec l'augmentation de la charge

3. Efficacité relative des différentes couches :

- La séparation lecture/écriture au niveau BD apporte le gain de performance le plus important
- Le load balancing du frontend apporte également une amélioration significative
- Le load balancing du backend a un impact plus modeste mais toujours positif

4. Analyse coût-bénéfice :

- L'augmentation des ressources système (3x plus de conteneurs) produit un gain de performance de plus de 65%
- Ce ratio favorable justifie pleinement l'investissement dans l'architecture distribuée

5. Points d'optimisation potentiels :

- Les logs indiquent que le master DB reste le composant le plus sollicité
- Une optimisation future pourrait inclure le partitionnement des données (sharding)

Comparaison avec les attentes théoriques :

Théoriquement, une multiplication par trois des ressources pourrait laisser espérer une amélioration de performance de 200%. Nos résultats montrent une amélioration de 67-75% avec l'implémentation initiale, et jusqu'à 213% avec PostgreSQL sous charge très élevée. Ces différences s'expliquent par :

1. Overhead de coordination :

- Les load balancers ajoutent une légère latence
- La réplication des données consomme des ressources

2. Limites de parallélisation :

- Certaines opérations restent séquentielles (écritures sur le master)
- Le routage intelligent ajoute une complexité supplémentaire

3. Facteurs limitants :

- La machine hôte possède des ressources partagées (CPU, disque, réseau)
- Des contentions peuvent apparaître malgré la distribution

Néanmoins, le gain de performance obtenu représente un excellent compromis entre l'investissement en ressources et l'amélioration des performances, particulièrement sous forte charge.

6.5 Démonstration pratique du fonctionnement

Pour valider le comportement correct de notre système distribué en conditions réelles, nous avons réalisé plusieurs démonstrations pratiques illustrant le routage intelligent et la réplication des données.

6.5.1 Démonstration du routage des requêtes de lecture (GET)

Pour démontrer la distribution des requêtes de lecture, nous avons exécuté une série de requêtes GET consécutives et observé la répartition entre les instances de base de données.

Script de test :

```
for i in {1..10}; do
  curl -X GET http://localhost/api/data
  echo ""
done
```

Résultats observés :

Comme le montre l'image, les requêtes GET sont effectivement distribuées entre les trois instances de base de données (db1, db2, db3). Cette répartition confirme le bon fonctionnement du load balancer et de l'algorithme `least_conn` configuré pour les lectures.

Points clés à noter :

- Chaque réponse inclut l'identifiant du serveur qui l'a traitée
- La distribution n'est pas parfaitement équilibrée (ce qui est attendu avec `least_conn`)
- Toutes les instances retournent des données cohérentes, confirmant la bonne réplication

```

[{"db_response":{"message":"Donn\u00e9e 'Test data' sauvegard\u00e9e sur DB 1"},"server_id":1,"status":"success"}]
zina@zina-Latitude-5490:~/Documents/projet$ curl -X GET "http://localhost:8084/data"
{"message":"Test data","origin":"DB1"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X GET "http://localhost:8084/data"
{"message":"Test data","origin":"DB1","server":"DB2"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X GET "http://localhost:8084/data"
{"message":"Test data","origin":"DB1","server":"DB3"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X GET "http://localhost:8084/data"
{"message":"Test data","origin":"DB1"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X GET "http://localhost:8084/data"
{"message":"Test data","origin":"DB1","server":"DB2"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X GET "http://localhost:8084/data"
{"message":"Test data","origin":"DB1","server":"DB3"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X GET "http://localhost:8084/data"
{"message":"Test data","origin":"DB1"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X GET "http://localhost:8084/data"
{"message":"Test data","origin":"DB1","server":"DB2"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X GET "http://localhost:8084/data"
{"message":"Test data","origin":"DB1","server":"DB3"}
zina@zina-Latitude-5490:~/Documents/projet$

```

FIGURE 6.2 – Distribution des requêtes GET entre les instances de base de données

6.5.2 Démonstration du routage des requêtes d'écriture (POST)

Pour vérifier le routage exclusif des écritures vers le master, nous avons exécuté une série de requêtes POST et analysé leur destination.

Script de test :

```

for i in {1..10}; do
  curl -X POST -H "Content-Type: application/json" \
    -d '{"value":"test-$i"}' \
    http://localhost/api/data
  echo ""
done

```

Résultats observés :

```

zina@zina-Latitude-5490:~/Documents/projet$ # Envoyer une requête POST (écriture)
curl -X POST "http://localhost:8081/save" -H "Content-Type: application/json" -d '{"message": "Test data"}'
{"db_response":{"message":"Donn\u00e9e 'Test data' sauvegard\u00e9e sur DB 1"},"server_id":1,"status":"success"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X POST "http://localhost:8081/save" -H "Content-Type: application/json" -d '{"message": "Test data"}'
{"db_response":{"message":"Donn\u00e9e 'Test data' sauvegard\u00e9e sur DB 1"},"server_id":2,"status":"success"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X POST "http://localhost:8081/save" -H "Content-Type: application/json" -d '{"message": "Test data"}'
{"db_response":{"message":"Donn\u00e9e 'Test data' sauvegard\u00e9e sur DB 1"},"server_id":3,"status":"success"}
zina@zina-Latitude-5490:~/Documents/projet$ curl -X POST "http://localhost:8081/save" -H "Content-Type: application/json" -d '{"message": "Test data"}'
{"db_response":{"message":"Donn\u00e9e 'Test data' sauvegard\u00e9e sur DB 1"},"server_id":1,"status":"success"}
zina@zina-Latitude-5490:~/Documents/projet$

```

FIGURE 6.3 – Routage des requêtes POST exclusivement vers l'instance master (db1)

L'image démontre clairement que toutes les requêtes POST sont systématiquement dirigées vers db1 (le master), conformément à notre configuration. Ceci confirme l'efficacité du routage intelligent basé sur le type de requête.

Points clés à noter :


- 100% des écritures sont traitées par db1 (master)
- Aucune requête d'écriture n'est dirigée vers les replicas
- Les identifiants des données créées sont séquentiels, confirmant leur traitement par une instance unique

6.5.3 Vérification de la réplication des données

Pour vérifier que les données écrites sur le master sont correctement répliquées vers les replicas, nous avons effectué une écriture suivie de lectures multiples sur différentes instances.

Séquence de test :

1. Exécution d'une requête POST pour créer une nouvelle donnée
2. Exécution de plusieurs requêtes GET pour vérifier la disponibilité de cette donnée sur toutes les instances



Résultats :

- La donnée créée était immédiatement visible lors des lectures suivantes
- Toutes les instances (db1, db2, db3) ont retourné la donnée nouvellement créée
- Les timestamps des données étaient identiques sur toutes les instances

Cette démonstration confirme que :

- Le mécanisme de réplication PostgreSQL fonctionne correctement
- La cohérence des données est maintenue entre le master et les replicas
- La réplication se fait avec une latence négligeable dans nos conditions de test

Défis rencontrés et solutions

Au cours de ce projet, nous avons fait face à plusieurs défis techniques qui ont nécessité des solutions adaptées. Cette section présente les principaux obstacles rencontrés et les approches mises en œuvre pour les surmonter.

7.1 Problèmes de réplication

Défi : Dans notre implémentation initiale, les données ne se répliquaient pas toujours correctement entre le master et les replicas, ce qui entraînait des incohérences.

Solution :

- Implémentation d'un mécanisme de réplication plus robuste avec système de retry
- Ajout d'un backoff exponentiel pour les tentatives ultérieures
- Logging détaillé des opérations de réplication pour faciliter le débogage
- Migration vers PostgreSQL avec ses mécanismes de réplication natifs bien plus fiables

Résultat : La fiabilité de la réplication s'est considérablement améliorée, avec une cohérence des données presque parfaite entre le master et les replicas.

7.2 Identification des réponses

Défi : Initialement, il était difficile d'identifier l'origine des réponses (quelle instance avait traité la requête), ce qui compliquait le débogage et la validation du bon fonctionnement du load balancing.

Solution :

- Ajout d'un identifiant unique à chaque instance (via des variables d'environnement)
- Inclusion systématique de cet identifiant dans toutes les réponses API
- Ajout d'en-têtes HTTP personnalisés pour suivre le routage à travers les couches
- Configuration de logs Nginx pour enregistrer l'instance cible


Résultat : Une traçabilité complète du chemin de chaque requête, facilitant la vérification du fonctionnement correct du load balancing et l'identification des problèmes potentiels.

7.3 Configuration de Nginx

Défi : La configuration initiale de Nginx ne dirigeait pas correctement les requêtes, en particulier pour la séparation des opérations de lecture et d'écriture au niveau de la base de données.

Solution :

- Configuration avancée utilisant les directives `if` pour distinguer les méthodes HTTP

- 
- Création de deux groupes de serveurs (upstreams) distincts pour les lectures et les écritures
 - Utilisation de l'algorithme `least_conn` pour les lectures
 - Ajustement des timeouts pour tenir compte des temps de traitement variables
 - Configuration de health checks pour éviter de diriger des requêtes vers des instances indisponibles

Résultat : Un routage intelligent et efficace des requêtes, avec une séparation claire des opérations de lecture et d'écriture et une distribution optimale de la charge.

Conclusion et perspectives

8.1 Synthèse des résultats

Notre projet a démontré l'efficacité d'une architecture distribuée avec load balancing pour améliorer les performances et la disponibilité d'un système multi-tiers. Les résultats clés incluent :

- Amélioration significative des performances sous charge (jusqu'à +67.3% de requêtes par seconde)
- Réduction du temps de réponse (jusqu'à -40.2% en moyenne)
- Augmentation spectaculaire du taux de transfert (jusqu'à +239.7%)
- Diminution drastique du taux d'erreur (de 3.25% à 0.05% sous forte charge)
- Capacité à gérer des pics de charge sans dégradation importante des performances
- Maintien du service même sous charge extrême (500 requêtes parallèles)

Les tests avec PostgreSQL ont particulièrement mis en évidence l'importance du load balancing pour optimiser l'utilisation des ressources de base de données et éviter les limitations de connexions.

8.2 Avantages de l'architecture proposée

L'architecture multi-tiers avec load balancing que nous avons implémentée présente plusieurs avantages majeurs :

1. **Haute disponibilité** : La réplication des instances à chaque niveau élimine les points uniques de défaillance.
2. **Scalabilité horizontale** : Possibilité d'ajouter facilement de nouvelles instances à n'importe quelle couche pour répondre à l'augmentation de la charge.
3. **Isolation des composants** : Chaque couche peut évoluer indépendamment sans affecter les autres.
4. **Optimisation des ressources** : Distribution efficace de la charge pour maximiser l'utilisation des ressources disponibles.
5. **Robustesse** : Capacité à absorber les pics de charge et à maintenir un service stable même en cas de défaillance partielle.
6. **Flexibilité** : Architecture adaptable à différents types d'applications et de charges de travail.

8.3 Améliorations futures possibles

Plusieurs pistes d'amélioration pourraient être explorées pour renforcer encore notre architecture :

1. **Sharding de la base de données** : Partitionnement horizontal des données pour répartir la charge d'écriture sur plusieurs instances master.

2. **Failover automatique** : Mise en place de mécanismes pour promouvoir automatiquement un replica en master en cas de défaillance.
3. **Autoscaling** : Ajout dynamique d'instances en fonction de la charge observée.
4. **Caching distribué** : Intégration d'une couche de cache (Redis, Memcached) pour réduire la charge sur les bases de données.
5. **Monitoring avancé** : Mise en place d'outils de surveillance plus sophistiqués pour anticiper les problèmes potentiels.
6. **Content Delivery Network (CDN)** : Intégration d'un CDN pour optimiser la distribution des contenus statiques.

8.4 Applications pratiques

Cette architecture peut être appliquée dans de nombreux contextes professionnels :

1. **Applications web à fort trafic** : Sites e-commerce, plateformes de médias sociaux, services de streaming.
2. **Services API publics** : APIs exposées à des partenaires ou au grand public nécessitant une haute disponibilité.
3. **Applications critiques d'entreprise** : Systèmes ERP, CRM, ou autres applications métier nécessitant une continuité de service.
4. **Applications IoT** : Traitement de grands volumes de données provenant d'appareils connectés.
5. **Plateformes SaaS** : Services logiciels nécessitant une scalabilité et une disponibilité élevées pour servir de multiples clients.

En conclusion, notre projet a démontré avec succès les bénéfices tangibles d'une architecture distribuée avec load balancing. Les résultats obtenus confirment que cette approche permet d'optimiser l'utilisation des ressources, d'améliorer les performances et d'augmenter la résilience du système, en particulier sous forte charge.

Références et ressources

- Microsoft Azure Documentation. (2023). What is Load Balancing ?. Disponible sur : <https://docs.microsoft.com/fr-fr/azure/load-balancer/load-balancer-overview>
- Load Balancing with NGINX : https://nginx.org/en/docs/http/load_balancing.html
- HAProxy Configuration Manual : <https://www.haproxy.org/download/2.7/doc/configuration.txt>
- Elastic Load Balancing (ELB) : <https://aws.amazon.com/elasticloadbalancing/>
- Redis as a Cache : <https://redis.io/docs/manual/client-side-caching/>
- Replication and Sharding in MySQL : <https://dev.mysql.com/doc/refman/8.0/en/replication.html>
- PostgreSQL Documentation - High Availability, Load Balancing, and Replication : <https://www.postgresql.org/docs/12/ha.html>
- Docker Documentation - Networking : <https://docs.docker.com/network/>
- Flask Documentation : <https://flask.palletsprojects.com/>

Note : Le code complet et les configurations détaillées sont disponibles sur nos référentiels GitHub : `simple-loadbalancer` (version avec simulation de base de données) et `postgres-loadbalancer` (version avec PostgreSQL).