Comparaison entre NoSQL et Blockchain en tant que bases de données distribuées

Yasser Benmansour
Département de génie informatique et génie logiciel
Polytechnique Montréal
Montréal, Canada
yasser.benmansour@polymtl.ca

Mohammed Ridha Ghoul
Département de génie informatique et génie logiciel
Polytechnique Montréal
Montréal, Canada
mohammed-ridha.ghoul@polymtl.ca

Victor Kim
Département de génie informatique et génie logiciel
Polytechnique Montréal
Montréal, Canada
victor.kim@polymtl.ca

Anis Youcef Dilmi
Département de génie informatique et génie logiciel
Polytechnique Montréal
Montréal, Canada
youcef.dilmi@polymtl.ca

***Abstrait* — Le but de cette expérience est de mesurer la performance de deux types de base de données. En utilisant une base de données *NoSQL* et en la comparant à une base de données *blockchain*. La latence choisie comme indicateur avec des charges de travail variées. La plateforme *Hyperledger Fabric* est utilisée pour la *blockchain* et *Redis* pour la base de données *NoSQL*. Des outils de mesure de performance adaptés, tels que *Hyperledger Caliper* pour la *blockchain* et *YCSB* pour *Redis*, sont utilisés pour mesurer les métriques telles que la latence, l'utilisation du *CPU* et l'utilisation de la *RAM*. Chaque type de base de données est installé sur une instance *EC2 AWS* et isolé dans des conteneurs *Docker*. L'expérience est réalisée avec un total de 1000 transactions pour chaque *workload* différent avec des ratios d'opérations *READ/WRITE* variés. Les résultats montrent que *Redis* est plus performant que *Fabric* et qu'il est plus efficace pour la mise à l'échelle, tandis que *Fabric* offre une meilleure sécurité et confidentialité des données grâce à son mécanisme de vérification et de cryptographie sophistiqué, c'est d'ailleurs ce qui le rend moins performant en termes de rapidité. En somme, cette expérience a permis de mesurer la performance de deux types de bases de données populaires dans des situations de charge de travail variées, fournissant des indications sur leurs avantages et leurs limites respectives en termes de performances et de sécurité.**

***Mots clés* — base de données, nosql, blockchain, hyperledger, fabric, Caliper, ycsb, benchmark, redis, comparaison, test de performance.**

I. Introduction

Dans le domaine de l'informatique, les données sont un élément crucial du monde numérique. Par le passé, les entreprises disposaient d'une seule application et d'un matériel physique privé, ce qui nécessitait seulement une base de données physique pour stocker toutes les données. Cependant, depuis quelques années, l'utilisation d'internet a connu une croissance exponentielle et de nombreuses entreprises ont commencé à proposer des services en ligne, ce qui a considérablement augmenté la quantité de données à gérer. Cela a révélé plusieurs problèmes, notamment des goulots d'étranglement dus à un trop grand nombre de requêtes envoyées à un seul serveur, ainsi que des problèmes de performance de l'application. La variété des types de données posait également un problème, car il était impossible de prévoir tous les types de données qui seraient transmis. En outre, les coûts liés au matériel physique étaient importants [1].

Pour s'adapter à la quantité de données, une nouvelle méthode de stockage a été conçue consistant à décomposer les données en plusieurs partitions et stocker chaque partition dans des emplacements physiques différents. Les différents serveurs travaillent sur leur partie de données [2].

Les bases de données distribuées permettent de gérer des quantités très importantes de données et peuvent facilement s'adapter en fonction des besoins. De plus, chaque serveur s'occupe d'une partie des données et les serveurs communiquent ensemble au besoin pour assembler les différentes partitions de données, ce qui réduit la charge et assure la disponibilité du système. Les partitions de données sont également dupliquées plusieurs fois pour assurer la fiabilité des données [3]. Les utilisateurs perçoivent l'ensemble des composantes comme un système cohérent.

L'objectif de cette étude consiste à présenter une évaluation comparative de la plateforme de *blockchain Hyperledger Fabric* et d'une base de données *NoSQL*, en décrivant leurs principales différences en se basant sur leur documentation et leurs propriétés respectives. Nous présenterons également la méthode utilisée pour déployer chaque technologie ainsi que les outils d'analyse utilisés pour obtenir les résultats de l'évaluation comparative des performances de chaque technologie. Enfin, nous comparerons les résultats des deux systèmes afin de justifier les différences observées et d'expliquer les limitations et obstacles rencontrés, tout en vérifiant la cohérence des résultats obtenus.

II. Context - Background

A. *Hyperledger Fabric*

Hyperledger Fabric est une plateforme de *blockchain* d'entreprise open source, dotée de fonctionnalités avancées telles que la haute confidentialité, la scalabilité et la flexibilité [4]. Elle offre une grande modularité, permettant aux développeurs de personnaliser et d'étendre les

fonctionnalités de la plateforme pour répondre aux besoins spécifiques de leur entreprise. De plus, la plateforme est hautement extensible et peut être intégrée à d'autres systèmes existants pour offrir une expérience de *blockchain* transparente et intégrée. *Fabric* prend également en charge des applications de chaîne d'approvisionnement complexes avec une gestion fine des autorisations et des accès, offrant ainsi aux entreprises un meilleur contrôle de leur chaîne d'approvisionnement [5]. Son cas d'utilisation principal n'est donc d'être base de données, dans cet article nous testerons les qualités de *Fabric* en tant que base de données, pour cette raison nous y feront référence en l'appelant le base de données.

Fabric utilise un réseau *blockchain* où plusieurs organisations peuvent transiger. En effet, cette technologie se démarque par la possibilité de créer des canaux privés entre certaines organisations faisant partie du réseau, ceci permet de tenir un *ledger* privé auxquels les autres organisations n'ont pas accès [6]. Cette plateforme vise à surpasser certaines contraintes d'autres technologies *blockchain* comme le faible débit et la grande latence grâce au consensus *Crash Fault Tolerant* (*CFT*) qui est basé sur le protocole *Raft.* De plus, une des caractéristiques de *Fabric* est que le consensus est modifiable en fonction du cas d'utilisation [6]. En résumé, *Fabric* est une solution de *blockchain* d'entreprise complète, dotée de fonctionnalités avancées, qui répond aux besoins spécifiques des entreprises en matière de sécurité, de scalabilité et de flexibilité.

B. Base de données NoSQL

Les bases de données *NoSQL* sont conçues pour résoudre les problèmes de scalabilité et de flexibilité des bases de données relationnelles traditionnelles. Elles sont adaptées aux applications Web à forte charge de trafic et peuvent stocker des données semi-structurées ou non structurées telles que des données de capteurs, de médias sociaux et de géolocalisation. Contrairement aux bases de données relationnelles, les bases de données *NoSQL* ne nécessitent pas de schémas rigides pour stocker différents formats de données. Elles offrent une haute disponibilité et une tolérance aux pannes en raison de leur conception distribuée sur des *clusters* de serveurs [7]. En somme, les bases de données *NoSQL* sont évolutives, distribuées et flexibles, répondant aux exigences des applications modernes.

C. Comparaison

Bien que les bases de données *NoSQL* et *Fabric* partagent des propriétés similaires telles que la haute disponibilité, la tolérance aux pannes et la capacité à gérer des charges de travail évolutives, elles sont conçues pour des cas d'utilisation différents. Les bases de données *NoSQL* sont conçues pour stocker et gérer des données semi-structurées ou non structurées, tandis que *Fabric* est une plateforme de *blockchain* d'entreprise conçue pour des applications de chaîne d'approvisionnement complexes. Les deux technologies offrent des ressources complètes et bien organisées en ligne. Cependant, leurs caractéristiques distinctes répondent aux besoins spécifiques de chaque technologie. En somme, même si elles ont des propriétés de base similaires, les bases de données *NoSQL* et *Fabric* sont utilisées pour des cas d'utilisation différents.

III. Expérimentation

Après avoir effectué une analyse approfondie du sujet et présenté toutes les informations concernant le contexte de notre expérience, nous nous concentrerons sur la seconde partie, qui est consacrée à la pratique. Plus précisément, nous allons évaluer les performances de deux outils que nous avons choisies pour notre étude : *Hyperledger Fabric* pour la technologie *blockchain* et *Redis* pour le système de gestion de base de données *NoSQL.*

A. Setup des expérimentations

1) Déploiement

Nous avons mené nos deux banc d'essai sur une instance *AWS* de type t2.large équipée d'un système d'exploitation *Ubuntu* et disposant de 2 vCPU et 8 GiB de RAM. Nos tests ont montré que la seule exigence nécessaire est d'avoir au moins 8 GiB de RAM pour garantir un déroulement sans problème des tests. Aussi nous avons augmenté légèrement l'espace de stockage par défaut de 8GiB vers 20GiB. Nous nous connectons à ce serveur distant à partir d'un terminal avec la commande SSH. Les deux bases de données à l'étude, *Fabric* et *Redis* seront déployées dans un conteneur *Docker.*

2) Workloads

Pour les expériences, plusieurs *workloads* ont été définis afin de tester les bases de données. Les *workloads* sont proposés par les chargés de laboratoire, 3 différents ont été testés, chacun avec une configuration différente incluant des paramètres tels que le nombre d'opérations (équivalent à 1000) et leur répartition entre les opérations suivantes: *READ* et *WRITE*. Chaque *workload* a un rapport R/W différent pour tester les différents contextes de requêtes.

TABLEAU I. Workloads

1000 transactions	***Opérations***	***%***
Workload A	*READ*	100
	WRITE	0
Workload B	*READ*	10
	WRITE	90
Workload C	*READ*	50
	WRITE	50

3) Hyperledger Caliper/Fabric

L'installation du *setup Fabric* et *Caliper* nécessite l'installation de plusieurs paquets : *docker, nodejs, build-essentials, npm,* l'entrepôt git *caliper-benchmarks, hyperledger/caliper-cli,* l'image *docker hyperledger/fabric-ccenv, fabric-client, fabric-ca-client, nvm* et *python*

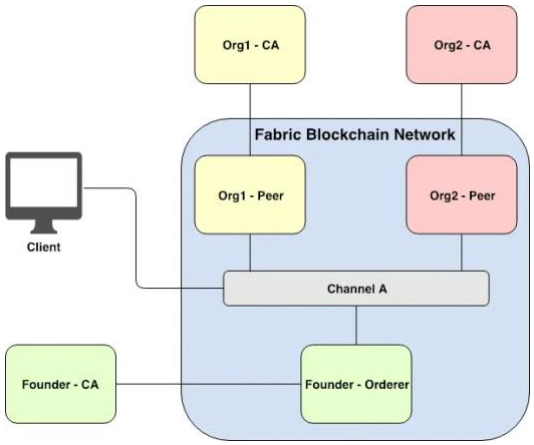

Fig. 1. Réseau blockchain de notre instance [8].

La figure 1 montre l'architecture de notre *blockchain Fabric*, nous pouvons en voir la description dans le fichier fabric-go-tls-solo.yaml. Chaque *peer* tient une version à jour du *ledger* qu'ils partagent. De plus notre instance de *Fabric* utilise *CouchDB* comme base de données d'état (*state DB*).

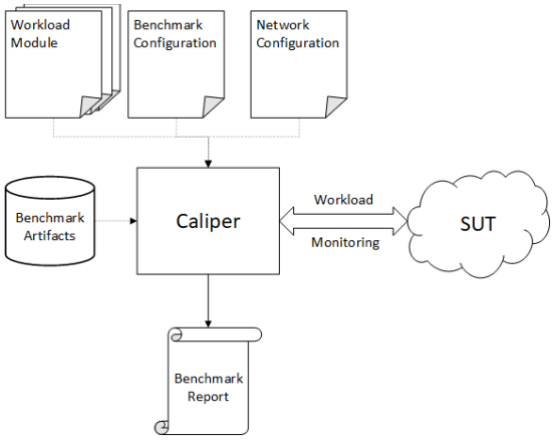

Fig. 2. Vue de haut niveau de Caliper.

La figure 2 montre le fonctionnement de *Caliper* d'un point de vue haut niveau. *Caliper* doit prendre en entrée un fichier qui décrit les *workloads*, un fichier qui décrit chaque *round*, un fichier qui décrit le réseau du *system under test* (*SUT*) ainsi que des artefacts. Ensuite *Caliper* peut exécuter les *workloads* et monitorer les métriques.

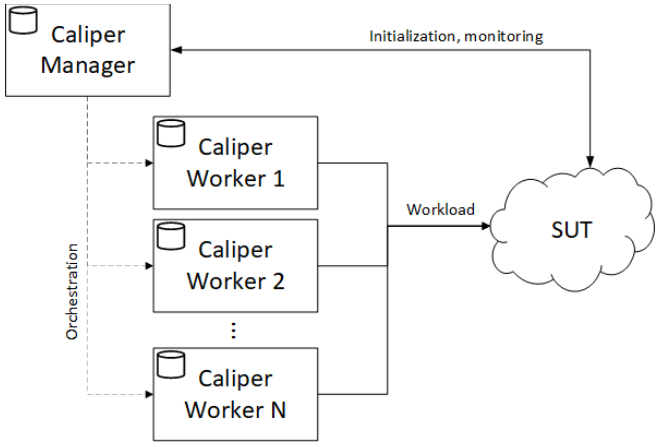

Fig. 3. Application des *workload* sur le SUT.

La figure 3 montre les processus de *Caliper*. Le *manager process* initialise le *SUT*, planifie les *rounds* et génère le rapport en fonction des performance observées. Les *worker processes* exécutent la charge de travail réelle sur le réseau *blockchain*. La figure 4 montre en détail le rôle d'un *worker*. Le *worker* exécute les round en respectant les paramètre du rate *controller*, en construisant les requêtes et en les soumettant au *SUT*. Le *worker* produit un rapport de la progression pour le *manager process*. Notre *setup* utilise 5 *workers*.

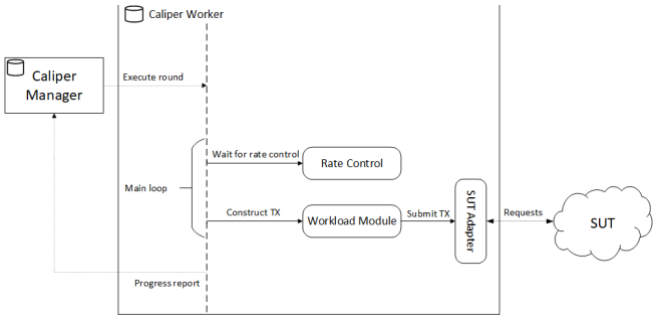

Fig. 4. Fonctionnement des *workers*.

Comme mentionné précédemment, pour évaluer les performances de *Fabric*, nous avons utilisé *Caliper*. Dans *Caliper*, nous avons modifier le fichier workload.js pour que les transactions envoyé sur *Fabric* soient un mélange de *read* et *write* tout en respectant la proportion désirée c.à.d. 50/50 *R/W*, 10/90 *R/W* et 100*R*, voir la figure 18 en annexe. Nous avons donc effectué 4 *rounds*, le premier pour populer le *blockchain* et les 3 autres pour faire les tests mentionnés plus haut, chaque round envoie 1000 transactions, voir le fichier config.yaml dans figure 21 en annexe. Pour démarrer le *benchmarking*, nous utilisons la commande :

```
sudo npx caliper launch manager
--caliper-workspace .
--caliper-benchconfig config.yaml
--caliper-networkconfig fabric-go-tls-solo.yaml
```

La première option sert à spécifier le *worskpace* utilisé pour faire les tests. La deuxième option spécifie le fichier de configuration pour le nombre de *rounds* ainsi que leurs paramètres (tx, tps). Finalement la troisième option spécifie le fichier des spécifications du *blockchain* sur lequel les tests vont être effectués.

Pour l'évaluation de la consommation de ressources, nous avons utilisé la commande *docker stats*. Nous avons créé un script pour que la commande ne monitor que les conteneurs suivants :

- couchdb.peer0.org1.example.com
- couchdb.peer0.org2.example.com
- ca.org1.example.com
- ca.org2.example.com
- orderer0.example.com
- peer0.org1.example.com
- peer0.org2.example.com

Pendant le monitorage, le script imprime la consommation dans un fichier texte à chaque 2 secondes. Ensuite nous obtenons en sortie un long fichier contenant la

consommation des conteneurs pendant l'intervalle de temps correspondant aux

4) YCSB/Redis

Nous utilisons un outil libre appelé *The Yahoo! Cloud Serving Benchmark* (*YCSB*) pour effectuer nos tests de performance. Développé par *Yahoo! Labs*, cet outil permet d'évaluer de façon standardisée les performances de différentes bases de données et systèmes de stockage. *YCSB* propose un ensemble de charges de travail pour mesurer la performance, l'évolutivité et la disponibilité des systèmes basés sur l'informatique en nuage, et il est généralement utilisé pour comparer les performances relatives des systèmes de gestion de base de données *NoSQL*. Pour mener notre expérience, nous avons suivi un script et un guide de démarrage publiés sur le dépôt *GitHub* du groupe de l'année dernière [9]. Ces étapes impliquaient l'installation des bibliothèques nécessaires, la création et la configuration de chaque environnement virtuel, et enfin l'exécution des tests.

Afin de tester la base de données *NoSQL Redis*, nous avons utilisé 4 nœuds au total, dont 1 primaire et 3 répliques. Pour reproduire l'étude, veuillez suivre les instructions dans le fichier *readme* qui se trouve dans le répertoire racine du dépôt, afin d'installer les dépendances nécessaires et exécuter le script. Les détails de la configuration des conteneurs sont spécifiés dans le script, que nous avons trouvé sur le *GitHub* du groupe de l'année dernière [9].

Pour faciliter la procédure d'évaluation des performances, nous avons opté pour l'utilisation d'un script *bash shell* que nous avons récupéré sur le *GitHub* du groupe de l'année précédente [9]. Avant d'exécuter le script, nous avons installé toutes les dépendances nécessaires en suivant les instructions détaillées dans le fichier *readme*. Le script installe *Git*, *python2* et *python3 virtualenv* pour assurer l'exécution complète du processus. Dans un premier temps, il crée un environnement virtuel à l'aide de *virtualenv* pour mener les tests, puis il clone le référentiel de référence et procède à un nettoyage de l'environnement.

Pour déployer la base de données dans l'environnement virtuel, nous avons opté pour les technologies *docker-compose* et *docker*. Nous allons créer un conteneur *docker* et lancer les tests *YCSB* après avoir terminé la configuration. Nous allons exécuter un total de trois tests pour déterminer une moyenne des valeurs de sortie en utilisant la stratégie Simple avec un facteur de réplication de 3, ce qui nécessitera un total de 4 nœuds pour répliquer l'environnement réseau. Une fois les trois tests terminés, un seul fichier de résultats sera automatiquement généré sous forme de fichier CSV dans le répertoire approprié. Enfin, le programme *docker-compose* se terminera et affichera un message de fin.

IV. RÉSULTATS

A. Graphiques des résultats

1) Workload A (100/0 R/W)

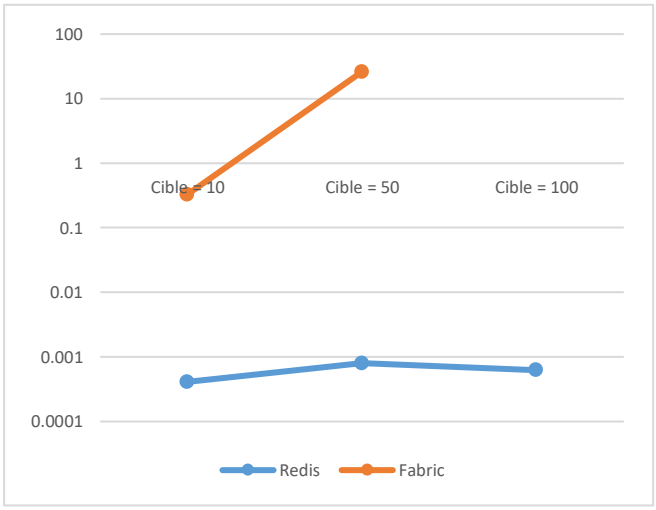

Fig. 5. Latence (s) 100/0 R/W

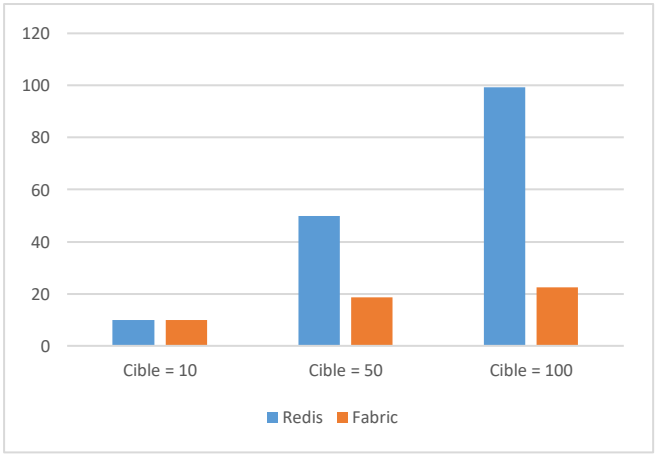

Fig. 6. Débit (ops/s) 100/0 R/W

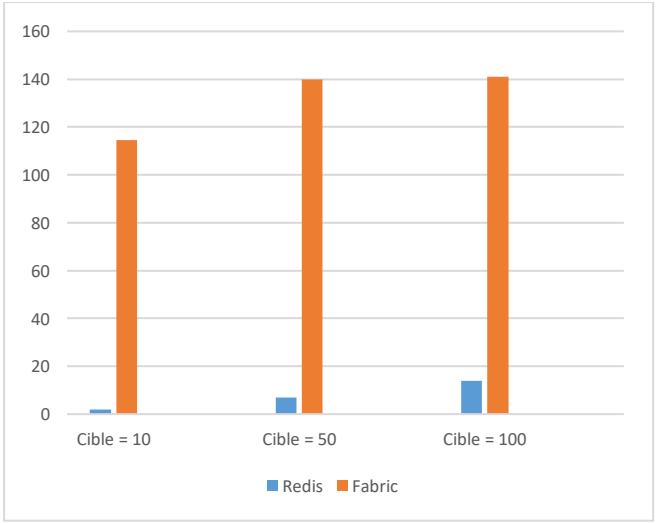

Fig. 7. CPU usage (%) 100/0 R/W

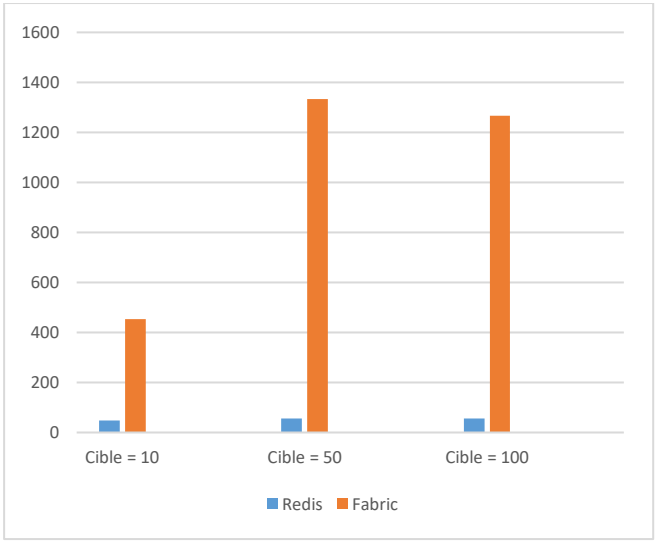

Fig. 8. RAM usage (MiB) 100/0 R/W

2) *Workload B (10/90 R/W)*

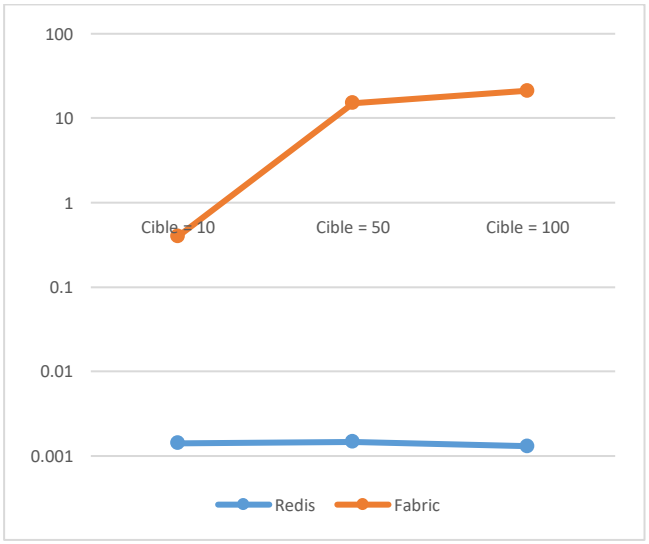

Fig. 9. Latence (s) 10/90 R/W

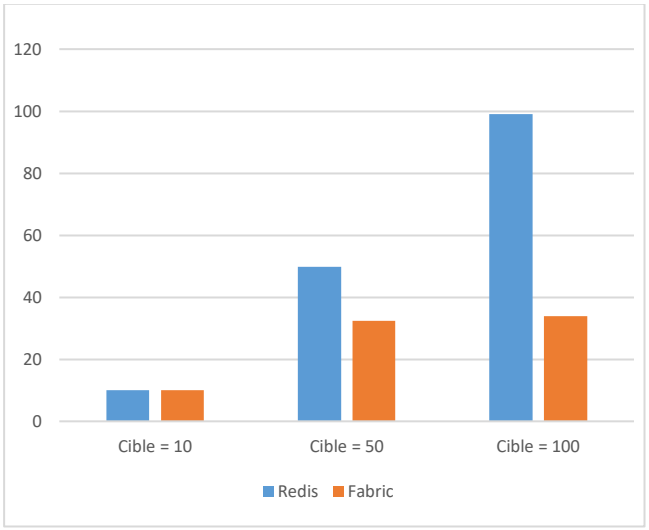

Fig. 10. Débit (ops/s) 10/90 R/W

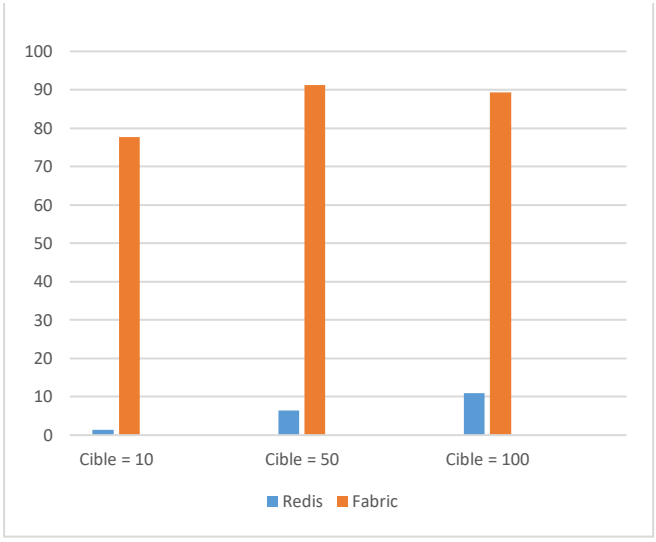

Fig. 11. CPU usage (%) 10/90 R/W

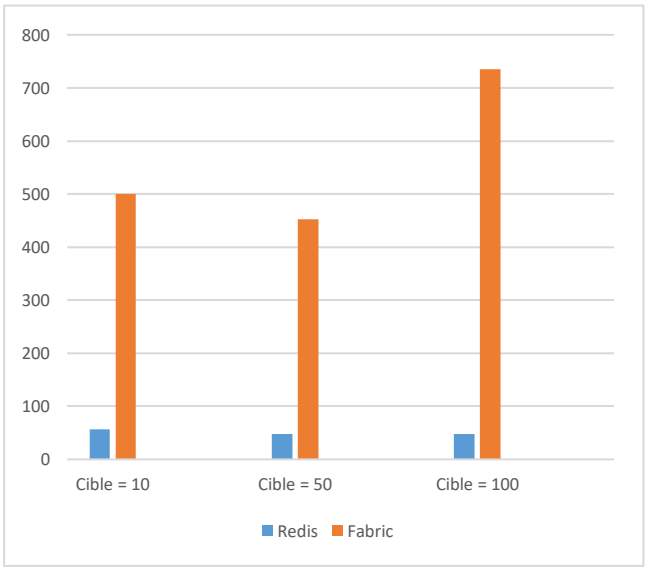

Fig. 12. RAM usage (MiB) 10/90 R/W

3) *Workload C (50/50 R/W)*

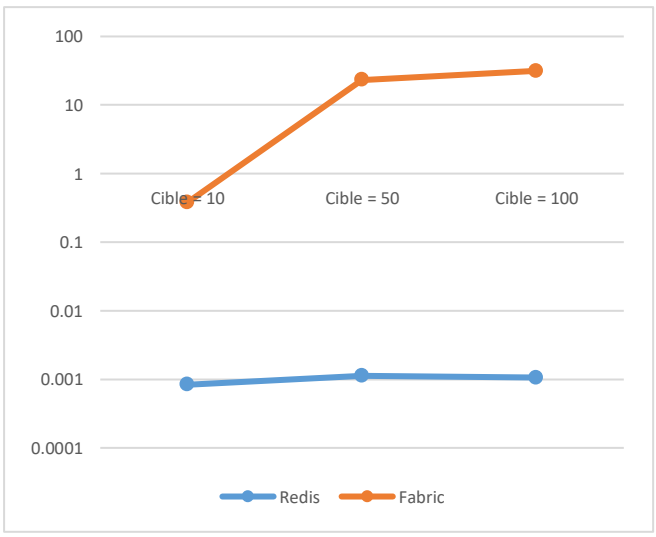

Fig. 13. Latence (s) 50/50 R/W

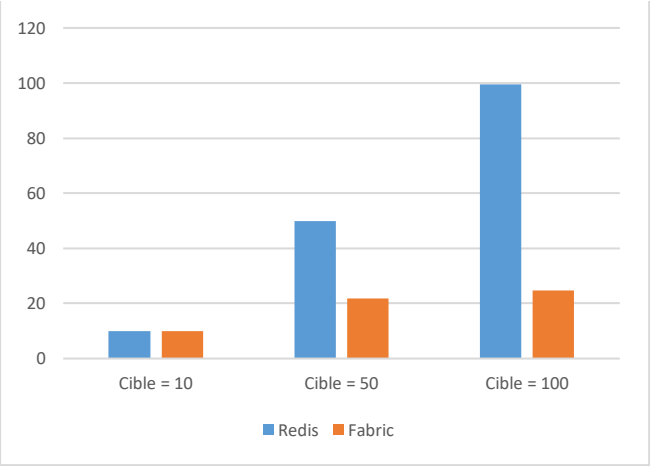

Fig. 14. Débit (ops/s) 50/50 R/W

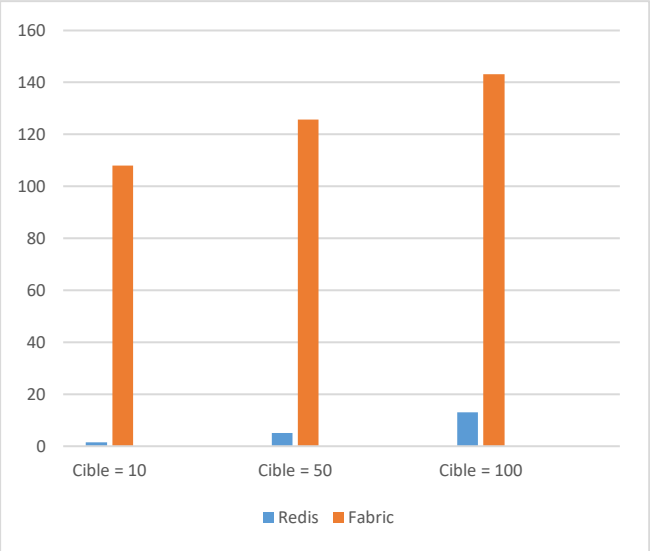

Fig. 15. CPU usage (%) 50/50 R/W

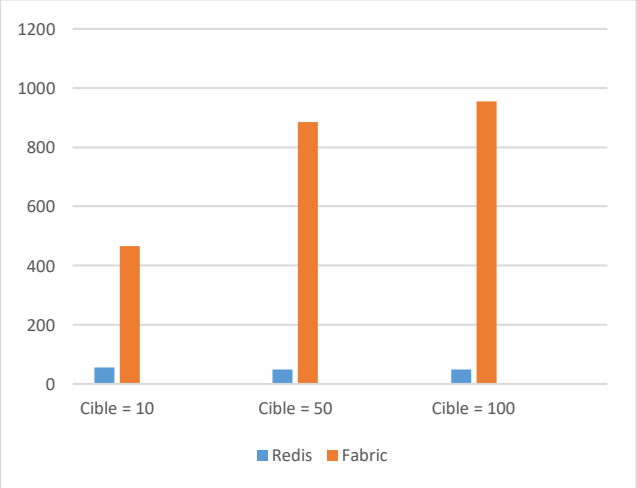

Fig. 16. RAM usage (MiB) 50/50 R/W

Pour alléger la lecture, nous avons mis en annexe les tableaux contenant les résultats sous forme numérique.

B. *Description des résultat*

Les résultats montrent que Redis consomme beaucoup moins de mémoire que *Fabric*, tandis que *Fabric* consomme plus de *CPU* que *Redis*. Ces différences peuvent être attribuées aux différences architecturales et fonctionnelles entre les deux systèmes. *Redis* est une base de données en mémoire qui stocke les données dans la RAM, et *Fabric* est un *framework blockchain* qui stocke les données sur disque. Cela peut expliquer pourquoi *Redis* utilise moins de mémoire mais nécessite une utilisation plus élevée du processeur. De plus, les différences de résultats peuvent également être attribuées aux différences architecturales et fonctionnelles entre les deux systèmes. Par exemple, *Redis* est optimisé pour la lecture des données, tandis que *Fabric* est conçu pour valider et vérifier les transactions blockchain. Cela peut entraîner des performances différentes en fonction de la charge de travail que nous exécutons. Pour les deux bases de données, les données de consommation de ressource ont été prélevées (*docker stats*) à des intervalles de 2 secondes pendant la période où les transactions étaient traitées, voir le script bash de la figure 19 en annexe. Ensuite nous avons fait la moyenne des prélèvements. Voir le script pythin de la figure 20 en annexe.

De plus, pour le *workload* A, cible 100 ops/s, Caliper n'a pas donné de valeur en sortie pour la latence car 100% des requêtes ont échoué.

En fin de compte, les différences observées dans les résultats des tests de performance peuvent être attribuées aux différences d'architecture et de fonctionnalités entre les deux systèmes. Nous devons considérer les avantages et les inconvénients de chaque système en fonction de leur cas d'utilisation spécifique pour déterminer celui qui convient le mieux à notre besoins

C. *Discussion*

En comparant les résultats des deux systèmes, nous pouvons observer que Redis est toujours plus performant que *Fabric* en termes de débit et de latence pour les *workloads* et les trois valeurs cibles (10, 50 et 100). *Redis* utilise également moins de *CPU* et *RAM* que *Fabric* pour tous des *workloads* et valeurs cibles. Par exemple, dans le *workload* A (100/0 R/W) à la cible 10, *Redis* a un débit de 9,98 ops/s et une utilisation moyenne de *RAM* de 114,6 MiB, tandis que *Fabric* a un débit de 10 ops/s et une utilisation moyenne de *RAM* de 453,68 MiB. De même, dans le *workload* B (50/50 R/W) à la cible 100, *Redis* a un débit de 99,46 ops/s et une latence de 0,001059 s, tandis que *Fabric* a un débit de 24,7 ops/s et une latence de 31,52 s.

Les différences entre les deux systèmes peuvent être attribuées à plusieurs facteurs, tels que leur architecture, les choix de conception et les détails de mise en œuvre. *Redis* est un magasin de données en mémoire qui utilise une architecture événementielle à un seul fil, ce qui lui permet d'atteindre un débit élevé et une faible latence. En revanche, *Fabric* est une plateforme de *ledger* distribué qui utilise une architecture modulaire et prend en charge plusieurs algorithmes de consensus, ce qui peut entraîner une complexité et une surcharge accrues. *Redis* prend également en charge le partitionnement et la réplication des données, ce qui peut améliorer l'évolutivité et la tolérance aux pannes, tandis que *Fabric* utilise une approche basée sur le consensus qui peut entraîner une latence plus élevée et un débit plus faible. De plus, nous avons remarqué que dans le *benchmark* de *Fabric*, plusieurs requêtes étaient cancellées par le système lorsque la cible était 50 ou 100 ops/s, c'est la raison pourquoi le débit était très faible. Le message d'erreur qui indiquait pourquoi les requêtes étaient

abandonnées disait : *too many open files*. Cela suggère que le processus *Fabric* (ou *CouchDB*) ouvre trop de fichiers simultanément, plus que le système ne le permet. Nous avons essayé d'augmenter la limite du système *Ubuntu* de notre instance *EC2* mais cela n'a pas fonctionné. Avant de valider une transaction, un système *blockchain* doit passer par plusieurs étapes qui incluent un consensus entre plusieurs nœuds, ces nœuds doivent communiquer entre eux et ceci ajoute au délai de la transaction.

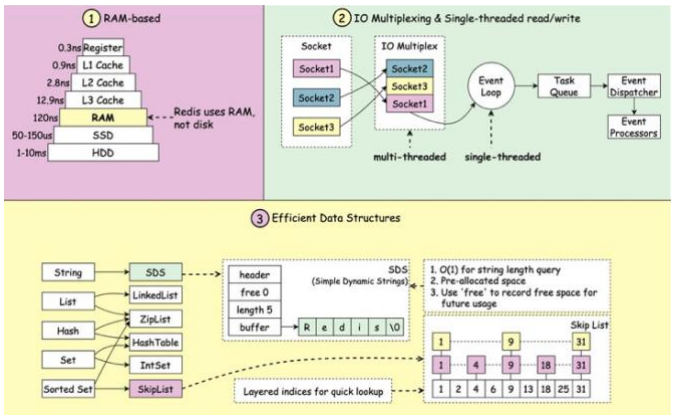

Fig. 17. Pourquoi Redis performe mieux [10].

Le travail présenté ici peut présenter certaines limites ou obstacles à la validité. Voici quelques-unes de ces limites : Les expériences ont été menées sur une seule machine, ce qui peut ne pas refléter les performances des systèmes dans un environnement distribué. Les expériences ont été menées à l'aide d'un ensemble spécifique de paramètres, et les résultats peuvent ne pas être généralisés à d'autres charges de travail ou configurations. Les expériences n'ont pris en compte que trois *workload* (A, B et C) et ne peuvent pas représenter l'ensemble des charges de travail possibles que les systèmes peuvent rencontrer dans la pratique. Les expériences ont été menées en utilisant une version spécifique de *Redis* et de *Fabric*, et les résultats ne peuvent pas s'appliquer à d'autres versions ou implémentations. Il convient donc d'être prudent lors de l'interprétation des résultats et de leur généralisation à d'autres scénarios. D'autres expériences et analyses peuvent être nécessaires pour valider les résultats et identifier les limites et les compromis de l'utilisation de *Redis* et *Fabric* dans différents contextes.

V. Conclusion

En conclusion, il est important de comprendre que *Redis* et *Hyperledger Fabric* sont deux systèmes différents, conçus pour répondre à des besoins spécifiques en matière de stockage et de traitement des données. La performance, la scalabilité, la sécurité et la confidentialité sont des éléments critiques pour les systèmes de stockage de données. *Redis* est un système rapide et facilement mis à l'échelle, mais avec des limites en matière de sécurité et de confidentialité des données. *Fabric*, en revanche, offre une sécurité élevée grâce à la validation multi-nœuds de chaque transaction, ainsi qu'une confidentialité élevée pour les participants autorisés, mais au prix d'une certaine lenteur et d'une complexité accrue en matière d'évolutivité. En fin de compte, le choix entre *Redis* et *Fabric* dépendra des besoins spécifiques de l'application en termes de sécurité, de confidentialité, de vitesse et d'évolutivité. Cependant *Redis* est optimisé pour la lecture des données, tandis que *Fabric* est conçu pour valider et vérifier les transactions *blockchain*. Cela peut expliquer les différences observées en fonction de la charge de travail exécutée. Finalement, il est important de considérer les avantages et les inconvénients de chaque système en fonction de leur cas d'utilisation spécifique pour garantir des performances optimales et une sécurité appropriée pour déterminer celui qui convient le mieux à nos besoins.

Pour les travaux de recherche futurs, il serait pertinent d'explorer d'autres systèmes de bases de données en mémoire et des *frameworks blockchain* afin d'évaluer leur performance en comparaison avec *Redis* et *Fabric*. Des charges de travail complexes et diversifiées devraient également être prises en considération.

Références

[1] N. Grover, A. Kumar, A. Mittal and R. Kumar, "Understanding the Big Data World," Pearson IT Certification, 2016.

[2] V. C. Emeakaroha, R. S. Montero, I. M. Llorente and I. Foster, "Big Data Processing," SpringerLink, 2013.

[3] M. S. Islam and K. Wang, "Data Replication," SpringerLink, 2018.

[4] A. Parashar and R. K. Tiwari, "Big Data Analysis and Processing," IEEE Traditional Data Systems, 2020.

[5] Hyperledger, "Hyperledger," https://www.hyperledger.org, Accessed on: Apr. 19, 2023.

[6] Hyperledger Fabric, "What is Hyperledger Fabric?," https://hyperledger-fabric.readthedocs.io/en/release-2.5/whatis.html, Accessed on: Apr. 19, 2023.

[7] MongoDB, "NoSQL Explained," https://www.mongodb.com/nosql-explained, Accessed on: Apr. 19, 2023.

[8] S. M. Al Hossain, "Demystifying Hyperledger Fabric 1.3: Fabric Architecture," Medium, 2019. [Online]. Available: https://medium.com/@serial_coder/demystifying-hyperledger-fabric-1-3-fabric-architecture-a2fdb587f6cb. [Accessed: Apr. 19, 2023].

[9] M. Bekhet, "DB-TP3-LOG8430E," GitHub, 2021. [Online]. Available: https://github.com/markbekhet/DB-TP3-LOG8430E. [Accessed: Apr. 19, 2023].

[10] A. Ubeyratne, "Twitter post," Twitter, Oct. 31, 2021. [Online]. Available: https://twitter.com/alexxubyte/status/1622999903894110214. [Accessed: Apr. 19, 2023].

VI. Annexe

TABLEAU II. Cible 10 ops/s

Cible = 10	1000 transactions	Redis	Fabric
Workload A 100/0 R/W	Débit (ops/s)	9.98	10
	Latence(s)	0.0004115	0.33
Workload C 10/90 R/W	Débit (ops/s)	9.98	10
	Latence(s)	0.001421	0.4
Workload B 50/50 R/W	Débit (ops/s)	9.98	10
	Latence(s)	0.0008331	0.38

TABLEAU III. Cible 50 ops/s

Cible = 50	1000 transactions	Redis	Fabric
Workload A 100/0 R/W	Débit (ops/s)	49.87	18.5
	Latence(s)	0.0008025	26.16
Workload C 10/90 R/W	Débit (ops/s)	49.76	32.4
	Latence(s)	0.001466	15.17
Workload B 50/50 R/W	Débit (ops/s)	49.86	21.7
	Latence(s)	0.00112338	23.34

TABLEAU IV. Cible 100 ops/s

Cible= 100	1000 transactions	Redis	Fabric
Workload A 100/0 R/W	Débit (ops/s)	99.196	22.5
	Latence(s)	0.0006289	-
Workload C 10/90 R/W	Débit (ops/s)	99.14	34
	Latence(s)	0.001308	21.18
Workload B 50/50 R/W	Débit (ops/s)	99.46	24.7
	Latence(s)	0.001059	31.52

TABLEAU V. Usage ressources (cible 10 ops/s)

	Fabric		Redis	
Cible = 10	CPU usage (%)	RAM usage (MiB)	CPU usage (%)	RAM usage (MiB)
Workload A 100/0 R/W	114.6	453.68	2	48
Workload C 10/90 R/W	77.68	500.82	1.4	56
Workload B 50/50 R/W	107.9	466.49	1.5	56

TABLEAU VI. Usage ressources (cible 50 ops/s)

	Fabric		Redis	
Cible = 50	CPU usage (%)	RAM usage (MiB)	CPU usage (%)	RAM usage (MiB)
Workload A 100/0 R/W	139.99	1333.97	7	56
Workload C 10/90 R/W	91.27	452.4	6.5	48
Workload B 50/50 R/W	125.81	886.01	5	48

TABLEAU VII. Usage ressources (cible 100 ops/s)

	Fabric		Redis	
Cible = 100	CPU usage (%)	RAM usage (MiB)	CPU usage (%)	RAM usage (MiB)
Workload A 100/0 R/W	141.02	1267.09	14	56
Workload C 10/90 R/W	89.25	735.29	11	48
Workload B 50/50 R/W	143.15	955.51	13	48

```
'use strict';

const { WorkloadModuleBase } = require('@hyperledger/caliper-core');

const owners = ['Alice', 'Bob', 'Claire', 'David'];

/**
 * Workload module for the benchmark round.
 */
class RW5050Workload extends WorkloadModuleBase {
    /**
     * Initializes the workload module instance.
     */
    constructor() {
        super();
        this.txIndex = 0;
    }

    /**
     * Assemble TXs for the round.
     * @return {Promise<TxStatus[]>}
     */
    async submitTransaction() {
        if (Math.random < 0.5) {
            this.txIndex++;
            let marbleName = 'marble_' + '1_' + this.txIndex.toString() + '_' + this.workerIndex.toString();
            let marbleColor = colors[this.txIndex % colors.length];
            let marbleSize = (((this.txIndex % 10) + 1) * 10).toString(); // [10, 100]
            let marbleOwner = owners[this.txIndex % owners.length];

            const args = {
                contractId: 'marbles',
                contractVersion: 'v1',
                contractFunction: 'initMarble',
                contractArguments: [marbleName, marbleColor, marbleSize, marbleOwner],
                timeout: 30
            };

            await this.sutAdapter.sendRequests(args);
        } else {
            this.txIndex++;
            let marbleOwner = owners[this.txIndex % owners.length];
            const args = {
                contractId: 'marbles',
                contractVersion: 'v1',
                contractFunction: 'queryMarblesByOwner',
                contractArguments: [marbleOwner],
                timeout: 120,
                readOnly: true
            };

            await this.sutAdapter.sendRequests(args);
        }
    }
}

/**
 * Create a new instance of the workload module.
 * @return {WorkloadModuleInterface}
 */
function createWorkloadModule() {
    return new RW5050Workload();
}

module.exports.createWorkloadModule = createWorkloadModule;
```

Fig. 18. Fichier workload.js modifié pour 50/50 R/W.

```bash
#!/bin/bash

while true;
    do echo -n $(docker stats couchdb.peer0.org1.example.com couchdb.peer0.org2.example.com
ca.org1.example.com ca.org2.example.com orderer0.example.com peer0.org1.example.com peer0.org2.example.com --
-no-stream --format "{{ json . }}")"+";
    sleep 2;
done >> output.txt
```

Fig. 19. Script bash pour prélever les données de consommation.

```python
import re
import json


pattern = re.compile(r'(\d+\.\d+|\d+)(?=MiB)')
with open('output.txt', 'r') as f:
    data = f.read()
groups = data.strip().split("+")
totals_mem = []
totals_cpu = []
for group in groups:
    objects = re.findall(r'{[^}]*}', group)
    objs = [json.loads(obj) for obj in objects]
    mem_values = [float(pattern.search(obj['MemUsage']).group()) for obj in objs]
    cpu_values = [float(obj['CPUPerc'].replace('%', '')) for obj in objs]

    total_mem = sum(mem_values)
    total_cpu = sum(cpu_values)

    totals_mem.append(total_mem)
    totals_cpu.append(total_cpu)

average_mem = sum(totals_mem) / len(totals_mem)
average_cpu = sum(totals_cpu) / len(totals_cpu)

print("Average memory usage for all groups:", average_mem)
print("Average cpu usage for all groups:", average_cpu)
```

Fig. 20. Script Python pour faire la moyenne des prélèvements de consommation de ressources.

```yaml
test:
  workers:
    type: local
    number: 5
  rounds:
    - label: init
      txNumber: 500
      rateControl:
        type: fixed-rate
        opts:
          tps: 25
      workload:
        module: benchmarks/samples/fabric/marbles/init.js
    - label: 100r
      txNumber: 1000
      rateControl:
        type: fixed-rate
        opts:
          tps: 10
      workload:
        module: benchmarks/samples/fabric/marbles/workload100r.js
    - label: 10r_90w
      txNumber: 1000
      rateControl:
        type: fixed-rate
        opts:
          tps: 10
      workload:
        module: benchmarks/samples/fabric/marbles/workload10r_90w.js
    - label: 50r_50w
      txNumber: 1000
      rateControl:
        type: fixed-rate
        opts:
          tps: 10
      workload:
        module: benchmarks/samples/fabric/marbles/workload50r_50w.js
```

Fig. 21. Fichier config.yaml modifié pour nos 4 *rounds*.