Optimisation des performances de Hyperledger Fabric: Détection précoce des conflits de concurrence de multiversion

par

Helmi TRABELSI

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE
M. Sc. A.

MONTRÉAL, LE 19 NOVEMBRE 2021

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. Kaiwen Zhang, Directeur de mémoire
Département de génie logiciel et des TI, École de technologie supérieure

M. Francis Bordeleau, Président du jury
Département de génie logiciel et des TI, École de technologie supérieure

M. Chamseddine Talhi, Membre du jury
Département de génie logiciel et des TI, École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 17 NOVEMBRE 2021

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Je tiens à exprimer mes vifs remerciements à tous ceux qui par leurs travaux, leurs idées, leurs présentations, leurs collaborations, ou leurs relectures ont participé de près ou de loin à la réalisation de ce mémoire.

Je tiens tout d'abord à remercier les membres du jury d'avoir accepté d'examiner mon mémoire ainsi que pour le temps qu'ils ont consacré pour la lecture attentive afin de m'adresser leurs remarques pertinentes qui permettent d'améliorer mon travail.

Ma plus profonde gratitude va à mon superviseur Mr Kaiwen Zhang pour sa disponibilité, son encadrement, ainsi que pour ses conseils judicieux qui ont contribué à l'accomplissement de ce mémoire.

Je tiens à exprimer ma très profonde gratitude à mes parents ainsi que toute ma famille à qui je suis profondément redevable pour leur confiance et leur sacrifice tout au long de mes études.

Enfin, je remercie tous mes amis et collègues du laboratoire de recherche *FUSÉE LAB* pour leur encouragement.

Optimisation des performances de Hyperledger Fabric: Détection précoce des conflits de concurrence de multiversion

Helmi TRABELSI

RÉSUMÉ

Hyperledger Fabric est un système de blockchain privée qui dispose d'un système hautement modulaire et extensible pour le déploiement de blockchains sans permission qui a un effet majeur sur un large éventail de secteurs. Contrairement aux systèmes blockchain traditionnels tels que Bitcoin et Ethereum, Hyperledger Fabric utilise le modèle EOV pour le traitement des transactions : les transactions soumises sont exécutées par le noeud endorseurs, ordonnées et groupées par les services de commande et validées par les noeuds de validation.

Cette architecture EOV pose un problème bien documenté qui est le conflit de concurrence de multiversions. Cela se produit lorsque deux transactions tentent d'écrire et de lire la même clé dans le grand livre en même temps. Les solutions existantes pour résoudre ce problème incluent l'élimination des blocs au profit du traitement des transactions en streaming, la réparation des conflits lors de la phase de commande et la fusion automatique des transactions en conflit à l'aide des techniques CRDT.

Dans ce mémoire, nous proposons une nouvelle solution appelée *Détection précoce des conflits MVCC*. Notre solution détecte les transactions en conflit à un stade précoce de l'exécution de la transaction au lieu de les traiter jusqu'à la phase de validation pour être abandonnée. L'avantage de notre solution est de détecter les conflits le plus tôt possible afin de minimiser la surcharge des transactions conflictuelles sur le réseau, ce qui réduit la latence des transactions de bout en bout et augmente le débit effectif du système. Nous avons réussi à faire les modifications nécessaires sur le code source de Hyperledger Fabric afin d'implémenter notre solution. Nous proposons trois solutions différentes pour réaliser la détection précoce qui diffère par la structure de données utilisée pour le stockage des clés ainsi que la politique de verrouillage. Nos résultats montrent que nos solutions fonctionnent mieux que la solution basic de Hyperledger Fabric. Notre meilleure solution est la méthode SyncMap qui améliore le débit effectif de 23% et réduit la latence de 80%.

Mots-clés: Hyperledger Fabric, MVCC, EOV

Optimizing Hyperledger Fabric Performance : Early Detection for Multiversion Concurrency Control Conflictsin Hyperledger Fabric

Helmi TRABELSI

ABSTRACT

Hyperledger Fabric is a popular permissionless blockchain system that features a highly modular and extensible system for deploying permissioned blockchains which are expected to have a major effect on a wide range of sectors. Unlike traditional blockchain systems such as Bitcoin and Ethereum, Hyperledger Fabric uses the EOV model for transaction processing : the submitted transactions are executed by the endorsing peer, ordered and batched by the ordering services, and validated by the validating peers. Due to this EOV workflow, a well-documented issue that arises is the multi-version concurrency control conflict. This happens when two transactions try to writes and read the same key in the ledger at the same time. Existing solutions to address this problem includes eliminating blocks in favor of streaming transactions, repairing conflicts during the ordering phase, and automatically merging the conflicting transactions using CRDT techniques.

In this paper, we propose a novel solution called *Early Detection for MVCC Conflicts*. Our solution detects the conflicting transactions at an early stage of the transaction execution instead of processing them until the validation phase to be aborted. The advantage of our solution is that it detects conflict as soon as possible to minimize the overhead of conflicting transaction on the network resulting in the reduction of the end-to-end transaction latency and the increase of the system's effective throughput. We have successfully implemented our solution in Hyperledger Fabric. We propose three different implementations which realize early detection. Our results show that our solutions all perform better than the baseline Fabric, with our best solution SyncMap which improves the goodput by up to 23% and reduces the latency by up to 80%.

Keywords: Hyperledger Fabric, MVCC, EOV

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ACL	Access Control List
API	Application Programming Interface
CAP	Consistency-Availability-Partition Tolerance
CRDT	Conflict Free Replicated Data
DCS	Consensus-Scale-Decentralized
EMVCC	Early Multiversion Concurrency Control
EOV	Execute-Order-Validate
FPGA	Field Programmable Gate Array
HLF	Hyperledger Fabric
LAB	Local Area Blockchain
MDBC	Monnaie Digitale de Banque Centrale
MSP	Membership Services Provider
MVCC	Multiversion Concurrency Control
TDEMVCC	Time to Detect MVCC
TDMVCC	Time to Detect EMVCC
TPS	Transaction Par Seconde
UTXO	Unspent Transaction Output
vCPU	Virtual Central Processing Unit
VSCC	Validation Système Chaincode

INTRODUCTION

Ces dernières années ont connu l'émergence de la technologie de chaîne de blocs grâce aux avantages offerts par l'utilisation de ce concept tel que la confidentialité, l'immutabilité et élimination des parties tiers. La blockchain à touché pas mal de secteurs tels que les finances et la santé qui ont commencé l'implémentation des prototypes ou des applications. En analysant les produits développés, les techniciens trouvent souvent des insuffisances au niveau des performances de réseaux de blockchain. Prenons l'exemple de Ethereum qui peut assurer un débit moyen de 30 transactions par seconde (TPS) qui est largement inférieure à une application réelle telle que VISA qui assure un débit de 1700 TPS. Ce problème d'optimisation des performances des réseaux de chaîne de blocs est l'objet de plusieurs travaux de recherche qui ne cesse pas d'augmenter ces derniers jours.

Parmi les systèmes de blockchain, on cite HLF qui est un système de chaîne de blocs privé qui permet le développement d'application décentralisée dans une variété de domaines. Comparé à Ethereum (Buterin, 2014) et Bitcoin (Nakamoto, 2008), HLF offre des performances supérieures, en raison de l'algorithme de consensus utilisé et du nombre réduit de noeuds sur le réseau par rapport aux chaînes de blocs publiques. En outre, Fabric fournit un cadre flexible pour la gestion des responsabilités entre les parties à l'aide du fournisseur de services d'appartenance appelé MSP (Membership Services Provider), qui est un composant abstrait permettant de conserver les identités et les rôles de tous les nœuds appartenant à la même organisation.

La plupart des plates-formes de chaîne de blocs existantes telles que Ethereum et HLF permettent des calculs complets de Turing en exécutant un contrat intelligent pour une transaction donnée (Zheng *et al.*, 2020). Pour garantir la cohérence à travers le réseau, les noeuds exécutent des transactions pour générer l'état suivant de la blockchain une fois que le contenu du prochain bloc est connu (quelles transactions exécutées et dans quel ordre), c'est ce qu'on appelle l'architecture Order-Execute (OX) (Androulaki *et al.*, 2018). Les faiblesses de ce modèle OX sont : l'exécution

séquentielle des transactions au sein de chaque bloc qui limite le débit du système. Ainsi, les contrats intelligents doivent être installés sur tous les noeuds du réseau afin qu'ils puissent calculer l'état suivant de la blockchain, ce qui peut causer des problèmes de confidentialité.

Par contre, HLF utilise un modèle Execute-Order-Validate (EOV), où les transactions sont exécutées dans les noeuds d'endossement pour générer les ensembles de lecture-écriture, y compris les versions des clés utilisées pour la simulation de la transaction, puis les transactions sont ordonnées en blocs par les services de commande, et à la fin validées et soumis dans le grand livre. L'architecture EOV surmonte les limitations du modèle OX en fournissant le parallélisme de l'exécution des transactions sur différents noeuds d'endossement. L'inconvénient du modèle EOV est qu'un verrou en lecture-écriture est utilisé pour synchroniser les phases d'exécution et de validation. Pour résoudre ce problème, HLF implémente un mécanisme de contrôle de lecture et écriture basé sur les versions (MVCC) afin de garantir la cohérence de la blockchain. Lors de la validation des transactions, les noeuds de validation comparent les versions des ensembles de lecture-écriture générés aux versions des clés du grand livre pour éviter que deux transactions tentent de modifier et/ou lire les mêmes paires clé/valeur en même temps. Une transaction sera rejetée si son jeu de lecture contient une ancienne version d'une clé. Si elle est déclarée échouée, le client doit la soumettre à nouveau.

Le problème de conflit MVCC est une limitation bien documentée de HLF, car il diminue le débit effectif du système (également appelé Goodput). En effet, les blocs peuvent contenir des transactions qui vont échouées, mais ils occupent de l'espace dans les blocs et consomment du temps et des ressources de calcule sur le réseau résultant la dégradation de ses performances. Ces transactions rejetées doivent également être réessayées par les clients en tant que nouvelles transactions, ce qui génère une charge supplémentaire sur le système. En pratique, le problème MVCC peut avoir un impact sur les performances de la blockchain. En effet, des études récentes

montrent que dans des scénarios réalistes tels que les registres électroniques de santé, 40% des transactions échouent en raison d'un conflit MVCC (Chacko, Mayer & Jacobsen, 2021).

Dans le but d'optimiser les performances de HLF, nous présentons une nouvelle approche pour résoudre le problème MVCC appelé *Détection précoce des conflits MVCC*. Nous introduisons un mécanisme de détection précoce de conflit qu'on appelle EMVCC qui vise à réduire le nombre de conflits entre les transactions, ce qui augmente le débit global du système et diminue la latence des transactions. L'avantage de notre solution par rapport aux travaux existants est que le conflit MVCC est détecté au premier contact de la transaction avec le réseau blockchain lors de l'endossement de la transaction permettant d'améliorer les performances du réseau.

En plus de ce travail qui est le volet principal de notre projet de recherche, nous avons fait une analyse de performances de la version originale de HLF dans un contexte de MDBC au cours d'un stage à la banque du Canada qui a été l'occasion de prise en contact et d'expérimentation du réseau HLF .

Dans le cadre de ce mémoire, nos contributions principales sont les suivantes :

- Nous proposons la formulation théorique du problème de détection précoce des transactions conflictuelles lors de la phase d'endossement au lieu l'approche traditionnelle de gestion des conflits MVCC lors de la phase de validation (Section 4.2).
- Nous proposons une nouvelle solution appelée Early Detection of MVCC Conflicts permettant la détection précoce de conflit MVCC par la technique mise en cache local dans les noeuds d'endossement (Section 3.2). Nous fournissons une analyse théorique de notre solution pour calculer la probabilité de détection précoce des conflits MVCC en fonction de paramètres tels que la politique d'approbation, le nombre d'organisations, etc. (Section 4.3)
- Nous présentons trois implémentations de référence de notre solution : SyncMap, Lock-Free et Mutex Lock, leur principale différence est la structure de données utilisée pour le stockage des informations. (Section 3.2.2)

- Nous évaluons notre solution et la comparons à la référence de HLF et nous faisons une analyse de sensibilité pour étudier l'impact de différents paramètres de réseau et de charge sur les performances (Chapitre 5)
- Nous présentons une étude de cas d'application sur le réseau HLF ou on fait l'analyse des performances d'une application de monnaie digitale de banque centrale afin de mieux comprendre le réseau HLF ainsi que les défis à résoudre pour ce système.

Dans ce qui suit, ce mémoire sera organisé de la manière suivante : nous commençons par l'introduction des quelques notions de base où on va définir les concepts clés permettant de bien comprendre le contenu de ce mémoire. En deuxième lieu, nous fournissons une revue de littérature à travers les-quelles nous parcourons les travaux de recherche optant à étudier et optimiser les performances de HLF. Dans le troisième chapitre, nous fournissons la formulation du problème et dans le quatrième nous présentons les détails de nos trois solutions proposées ainsi qu'une analyse théorique en fonction des différentes politiques d'approbation. Nous consacrons le sixième chapitre pour l'évaluation et la comparaison de nos solutions avec la référence de HLF ainsi qu'une discussion des résultats obtenus. Le septième chapitre de ce mémoire présente un cas d'utilisation du réseau HLF dans le secteur financier qui est l'implémentation d'une monnaie digitale de banque centrale élaborée au cours d'un stage à la banque du Canada. Dans le dernier chapitre, nous récapitulons le parcours de deux années de ce mémoire ainsi que les principaux défis rencontrés dans le but de l'élaboration de ce travail.

CHAPITRE 1

NOTIONS DE BASE

Souvent les gens confondent les systèmes de chaînes de blocs et les cryptomonnaies, la chaîne de blocs est la technologie qui soutient les cryptomonnaies telles que Bitcoin et Ethereum. Vu le potentiel de cette technologie d'autres applications que les cryptomonnaies ont été introduites avec l'implémentation du concept du contrat intelligent. Ce concept est appelé Blockchain 2.0. Dans ce chapitre, nous allons introduire les chaînes de blocs ainsi que leur différent type. Ensuite nous allons comparer les différents types de systèmes de chaîne de blocs et enfin nous présentant les théorèmes CAP (Gilbert & Lynch, 2012) et DCS (Slepak & Petrova, 2018) qui permettent de comprendre comment choisir un système de chaînes de blocs ainsi que les limites imposées.

1.1 Chaîne de blocs

La chaîne de blocs est une sorte de base de données distribuée permettant le stockage permanent et immuable des blocs de transactions d'une manière chaînées assurant la robustesse de cette technologie. Tous les noeuds du réseau blockchain sont responsables de maintenir et sécurisés le registre partagé. Ainsi toutes altération ou changement de ce registre doit être approuvé par l'ensemble des participants en utilisant un consensus tel que la preuve de travail qui permet de se mettre d'accord sur la structure du prochain bloc à ajouter à la chaîne de blocs. Comme le montre la Figure 1.1, un système de chaîne de blocs est composé essentiellement de deux composantes. La première est la structure de chaîne de blocs ou chaque bloc contient un ensemble de transactions, un hash du bloc pour conserver l'intégrité de l'information et le hash du bloc précèdent qui joue le rôle d'une référence pour assurer le chaînage entre les blocs. La deuxième composante est le réseau des paires qui forme les réseaux pairs à pairs du système de chaîne de blocs. Ces machines répliquent la même chaîne de blocs et ils peuvent communiquer et échanger des informations pour garantir la cohérence du registre.

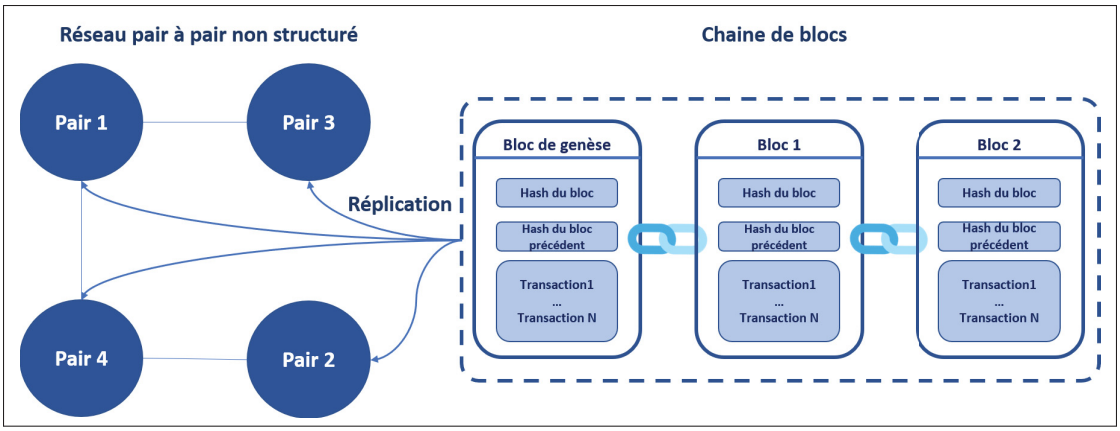


Figure 1.1 Structure des chaînes de blocs

1.1.1 Types des systèmes de chaîne de bloc

Il existe deux types de systèmes de chaîne de blocs :

- **Publique** : une chaîne de blocs publique est une chaîne de blocs décentralisée, sans autorité de gouvernance et ouverte au public, n'importe qui peut soumettre des transactions sur le réseau ou détenir le pouvoir de créer et valider des blocs en respectant l'algorithme de consensus. Toutes les données des transactions sur un tel réseau sont publiques.
- **Privée** : les chaînes de blocs privées sont des systèmes à permissions qui n'offrent pas le degré de décentralisation des chaînes de blocs publiques. Les participants au réseau sont bien définis et ils ne peuvent lire ou écrire sur le registre partagé que s'ils sont autorisés à le faire. Les données ne sont pas publiques et il y a aussi des techniques qui permettent d'assurer la confidentialité au sein du même réseau de chaîne de bloc tel que les collections de données privées dans HLF (Section 6.3.5). Les chaînes de blocs privées permettent de sécuriser l'interaction entre un groupe d'entités ayant le même objectif, mais qui ne font pas confiance entre eux.

Tableau 1.1 Comparaison entre les deux types de chaîne de blocs

	Publique	**Privée**
Accés	N'importe qui	Participant pré-sélectionné
Autorité	Décentralisé	Partiellement décentralisée
Débit de transaction	Faible	Élevé
Cout de transaction	Élevé	Faible
Données	N'importe qui peut lire et écrire	Seulement les organisations autorisées
Identité	Participant non identifiable	Participant identifiable

1.1.2 Comparaison entre les deux types de chaînes de blocs

En analysant les besoins fonctionnels de l'application qu'on veut développer sur un réseau de chaîne de blocs, on peut choisir entre les deux types du système qui diffèrent selon plusieurs caractéristiques :

- Accès : dans les réseaux privés, seulement les participants présélectionnés peuvent accéder au réseau. Par contre, pour les chaînes de blocs publiques il n'y a pas de restriction pour la participation, n'importe qui peut rejoindre le réseau à n'importe quel moment. En plus, il peut accéder aux données du registre et participer au processus de consensus.
- Autorité : dans des réseaux publics, on parle d'un niveau de sécurité due à la décentralisation au vrai sens du mot, aucune des entités du réseau n'a le pouvoir autoritaire pour prendre des décisions sans l'approbation des autres participants. Par suite, il est très difficile (mais pas impossible théoriquement) pour un noeud malicieux d'attaquer le système pour gagner son contrôle. Ceci est un peu différent dans le cas des réseaux privés où il y a une autorité de contrôle du réseau, en plus le réseau est partiellement décentralisé vu que le nombre de participants est limité aux intervenants présélectionnés.
- Débit de transaction : parmi les critères de choix les plus importants, on trouve le débit de transaction. En effet pour les blockchain publics le débit de transaction est relativement plus faible que les systèmes privés. Prenons l'exemple de Bitcoin qui assure un débit moyen de 7 TPS ou Ethereum ayant un débit de 20 à 30 TPS, mais si on parle de HLF on peut aller

à jusqu'à 1000 TPS ce qui rend les systèmes privés plus adaptés au cas d'utilisation des entreprises.

- Coût de transaction : le coût des transactions sur les réseaux publics est relativement élevé en comparant au réseau privé, ceci est dû à l'algorithme de consensus utilisé. Dans un réseau public, les mineurs doivent être récompensés pour maintenir le réseau et continuer à valider les transactions ce qui augmente les coûts des transactions alors que dans un réseau privé les participants sont bien connus et c'est leurs responsabilités de maintenir le réseau d'où le coût relativement faible des transactions sur ce type de chaînes de blocs.
- Données : dans les chaînes de blocs publics n'importe qui peut lire et écrire sur le grand livre mais ne peut pas faire une modification une fois la transaction est incluse dans un bloc ajouté à la chaîne. Mais dans une blockchain privée, les droits d'écriture et de lecture sont réservés au membre du réseau et peuvent être contrôlés par des ACL (Access-control list).
- Identité : la gestion des identités et la possibilité de restreindre l'accès aux données et aux transactions permettent aux administrateurs de blockchain privé de contrôler qui voit quel type d'informations dans quelles circonstances et de contrôler qui peut écrire ces informations sur la blockchain ce qui les rend idéales pour les opérations d'entreprise. Les réseaux publics assurent l'anonymat des participants ce qui fait que le système ne peut pas identifier un utilisateur en cas de fraude.

1.1.3 Théorème CAP

Le théorème CAP concerne les systèmes distribués en générale, en effet il permet de choisir le meilleur système distribué selon trois critères : cohérence (Consistency), disponibilité (Availability) et tolérance au partitionnement (Partition Tolérance). Le théorème CAP affirme qu'un système distribué ne peut fournir que deux de ces trois caractéristiques :

- **Cohérence** : elle signifie que les requêtes de lecture retournent toujours la valeur la plus récente ou une erreur, qui fait que tous les clients ont la même vue de données en même temps, quel que soit le nœud auquel ils seraient connectés.

- **Disponibilité** : les requêtes doivent être exécutées avec succès même si un ou plusieurs nœuds sont en panne.
- **Tolérance au partitionnement** : le système distribué doit continuer à fonctionner correctement malgré le nombre arbitraire de pannes de communication.

1.1.4 Théorème DCS

Le théorème CAP ne permet pas de différencier les systèmes de chaîne de blocs vue qui vont faire partie de la même classe. Le théorème DCS est plus granulaire permettant de classifier les systèmes de chaînes blocs, il est dans le même sens où un tel système ne peut fournir que deux sur trois des caractéristiques suivantes :

- **Cohérence** : la cohérence inclut la validité de l'exécution des transactions à travers un contrat intelligent sans avoir de problème d'incohérence telle que la double dépense. Elle englobe aussi la finalité des données dans le sens où l'information reste sur la chaîne de bloc à tout jamais ainsi que l'ordre des transactions qui fait que l'ensemble des transactions est ordonné dans des blocs qui sont enchaînés l'un après l'autre permettant d'obtenir un ordre total des transactions.
- **Décentralisation** : cette caractéristique permet de mesuré à quel point le système de chaîne de blocs est décentralisé dans le sens où le réseau est publique et il n'y a de tierce partie de confiance qui contrôle les décisions prises concernant la validité et l'ordre des transactions.
- **Scalabilité** : un réseau ayant une meilleure scalabilité est un réseau qui peut fonctionner avec un grand nombre de participants et peut traiter un volume important de transactions.

On peut citer l'exemple de Bitcoin qui assure la décentralisation ainsi que la cohérence des données, mais pas la scalabilité parce que le système offre un débit moyen de 7 transactions par second (TPS). Alors que HLF offre la scalabilité et la cohérence, mais il sacrifie la décentralisation puisqu'il s'agit d'un système fermé.

1.2 Survol de HLF

Dans cette section, nous allons parcourir les principaux composants de l'architecture HLF ainsi nous détaillons le flux de la transaction. Ensuite, nous expliquons brièvement le problème de conflit MVCC.

1.2.1 Architecture

Le réseau HLF est composé de plusieurs entités indispensables à son fonctionnement, citons les pairs qui peuvent avoir le rôle de pair endosseur ou validateur de transaction, les services de commande, les MSPs (Membership Services Provider), les clients et les canaux de communication.

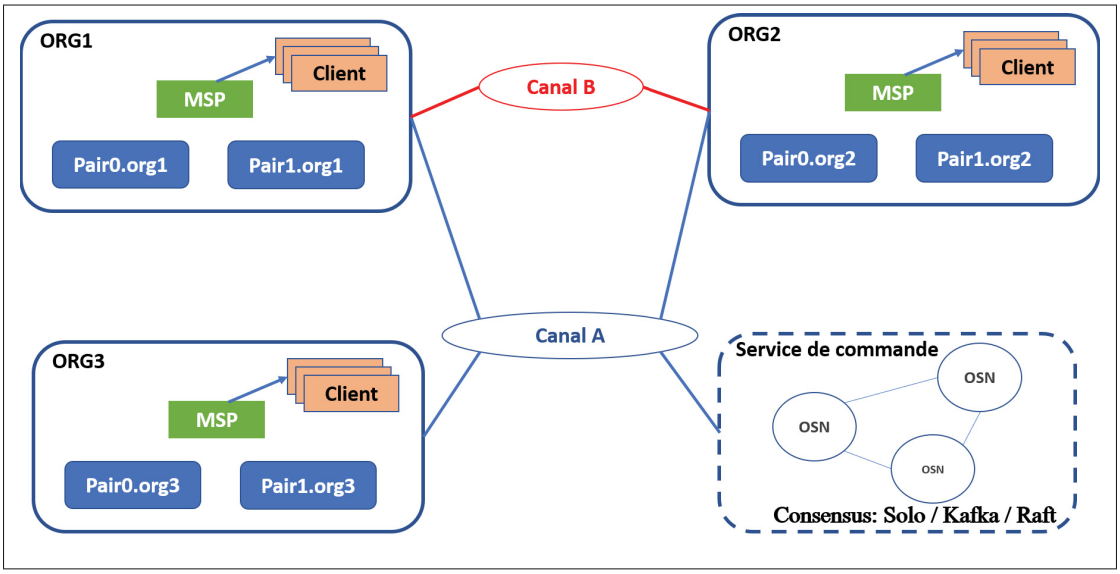


Figure 1.2 Exemple d'architecture d'un réseau HLF

1.2.1.1 Les pairs

Les pairs sont des composants fondamentaux du réseau HLF puisqu'il ils enregistrent l'état du réseau ainsi qu'une copie du grand livre. Chaque pair sur le réseau appartient à une organisation qui peut avoir un ou plusieurs nœuds. La Figure 1.2 illustre un exemple de réseau constitué de

trois organisations chacune à deux pairs. Pour participer au réseau et exécuter des transactions, le nœud doit demander et détenir un certificat qui lui permet de rejoindre le canal partagé entre tous les participants. Il existe deux types de pairs, les pairs d'endossement et les pairs de validation. Un pair sur le réseau peut être à la fois endosseur et validateur ou il peut jouer seulement le rôle d'un validateur, dans ce cas il n'est pas obligatoire d'installer le chaincode sur ce pair. Chaque pair a un administrateur qui peut créer, démarrer, arrêter ou reconfigurer le pair.

1.2.1.1.1 Les pairs d'endossement

Ce type de nœud héberge le contrat intelligent permettant de simuler les transactions, générer les ensembles de lecture/écriture et approuver les réponses de proposition de transaction à l'aide de la signature du pair. Lors de l'installation du chaincode, l'administrateur précise quels pairs peuvent HLF les transactions en définissant politiques d'approbation (Section 1.2.2).

1.2.1.1.2 Les pairs de validation

Les pairs de validations ont le rôle de vérification de la validité des transactions. Ils appliquent un processus de validation tel que illustré par la Figure 1.3 :

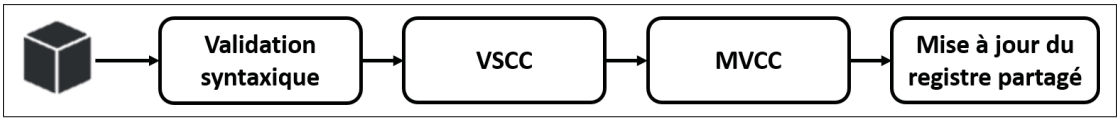


Figure 1.3 Pipeline de validation d'un bloc de transaction

1. Validation syntaxique : cette étape consiste à faire la vérification syntaxique de la structure globale ainsi que la structure des transactions contenue dans le bloc.
2. Validation du système du chaincode (VSCC) : consiste à valider les politiques d'endossement, en vérifiant que la transaction satisfait la politique d'endossement du chaincode. Une transaction qui n'a pas eu les endossements adéquats sera rejetée.
3. Validation MultiVersion Concurrency Control (MVCC) : cette étape est utile pour s'assurer que la version des clés lues pendant la phase d'endossement est toujours la même dans le

grand livre au moment de la validation pour garantir le déterminisme de l'exécution des transactions.

4. Soumission de transaction au grand livre : les transactions validées vont être soumises au grand livre partagé pour mettre à jour les clés correspondantes.

1.2.1.2 Les services de commande

Ce service peut être formé par un ou plusieurs noeuds de commande exécutant un algorithme de consensus déterministe qui peut être Solo, KafKa ou Raft. Son rôle principal est la collecte et l'ordonnancement des transactions ainsi que la formation des blocs. Le nombre des transactions par bloc dépend de la configuration du canal qui permet de définir la taille maximale du bloc ainsi que la durée qui doit être écoulée pour générer le nouveau bloc (BatchSize et BatchTimeout). En plus de leur rôle d'ordonnancement des transactions, les noeuds de commandes maintiennent une liste des organisations autorisées a créer des canaux qui peut être modifiée par l'administrateur du service de commande. Ce service applique également un contrôle d'accès aux canaux, définissant qui peut lire et écrire les données des canaux ou les configurer.

1.2.1.3 Fournisseur de services d'appartenance (MSP)

C'est un composant qui offre une abstraction des opérations d'adhésion. Un MSP cache derrière toutes les opérations d'émission et validation des certificats ainsi que l'authentification des utilisateurs. Un réseau HLF peut être gouverné par un ou plusieurs MSP selon le choix et les besoins fonctionnels de l'application. Sur le réseau illustré par la Figure 1.2, chaque organisation met en place son MSP pour gérer les identités de ses utilisateurs.

1.2.1.4 Client

Les clients sont les applications ou les logiciels qui jouent le rôle d'un utilisateur pour soumettre des transactions sur le réseau Fabric via le kit de développement de HLF.

1.2.1.5 Canaux de communication

Un canal HLF est un « sous-réseau » privé de communication entre deux ou plusieurs membres spécifiques du réseau, dans le but d'effectuer des transactions privées et confidentielles. Un réseau HLF peut avoir un ou plusieurs canaux que les pairs autorisés par les MSPs peuvent les rejoindre. Chaque transaction est exécutée sur un canal où les différentes parties prenantes doivent être authentifiées et autorisées à soumettre des transactions. La Figure 1.2 représente un exemple d'un réseau HLF en utilisant deux canaux. Le canal A est le canal public du réseau où les trois organisations peuvent soumettre des transactions alors que le canal B est partagé entre les deux organisations Org1 et Org2 seulement par contre Org3 ne peut pas consulter les données sur ce canal.

1.2.2 Politiques d'approbation

HLF permet aux développeurs de définir des politiques au niveau du chaincode. Les politiques d'endossement sont des règles qui spécifient quels pairs/organisations peuvent approuver l'exécution des transactions avant qu'elle ne soit appliquée au grand livre. En général, les politiques d'endossement sont configurées lors de l'installation du chaincode et ne peuvent être modifiées que lors d'une mise à niveau du contrat intelligent. Une fois le client crée la transaction, il envoie une proposition de transaction à tous les pairs endosseurs (Manevich, Barger & Tock, 2018) et attend les réponses à sa proposition. Lorsque le client reçoit suffisamment de réponses et de signatures pour satisfaire à la politique d'endossement dû chaincode, il peut soumettre la transaction avec les signatures d'endossement aux services de commande. Ces politiques d'endossement peuvent être définies comme suit :

- `AND('Org1.member','Org2.member')` : le client a besoin d'une signature de chaque organisation pour pouvoir soumettre la transaction aux services de commande.
- `OR('Org1.member','Org2.member')` : le client peut soumettre la transaction s'il reçoit un endossement de la part d'une des deux organisations.

- `OR('Org1.member',AND('Org2.member' ,'Org3.member'))` : afin de soumettre la transaction, le client doit collecter soit un endossement de la part d'un pair de l'organisation1 soit deux endossements, un de la part d'un pair de l'organisation2 et un de la part d'un pair de l'organisation3.
- `Out-Of(2,'Org1.member', 'Org2.member','Org3.member')` : au minimum deux des trois organisations endorsent la transaction pour qu'elle soit valide. Cette politique d'approbation est équivalente à `OR(AND('Org1.member','Org2.member') ,AND('Org1.member','Org3.member'), AND('Org2.member', 'Org3.member'))`

1.2.3 Fonctionnement d'une transaction

La Figure 1.4 montre le flux de transaction de base sur un réseau HLF. Comme on à déjà mentionné, HLF utilise le modèle EOV ou la transaction est en premier exécutée ensuite elle passe à l'ordonnancement et elle fini par la validation. On peut décomposer l'exécution de la transaction en cinq étapes :

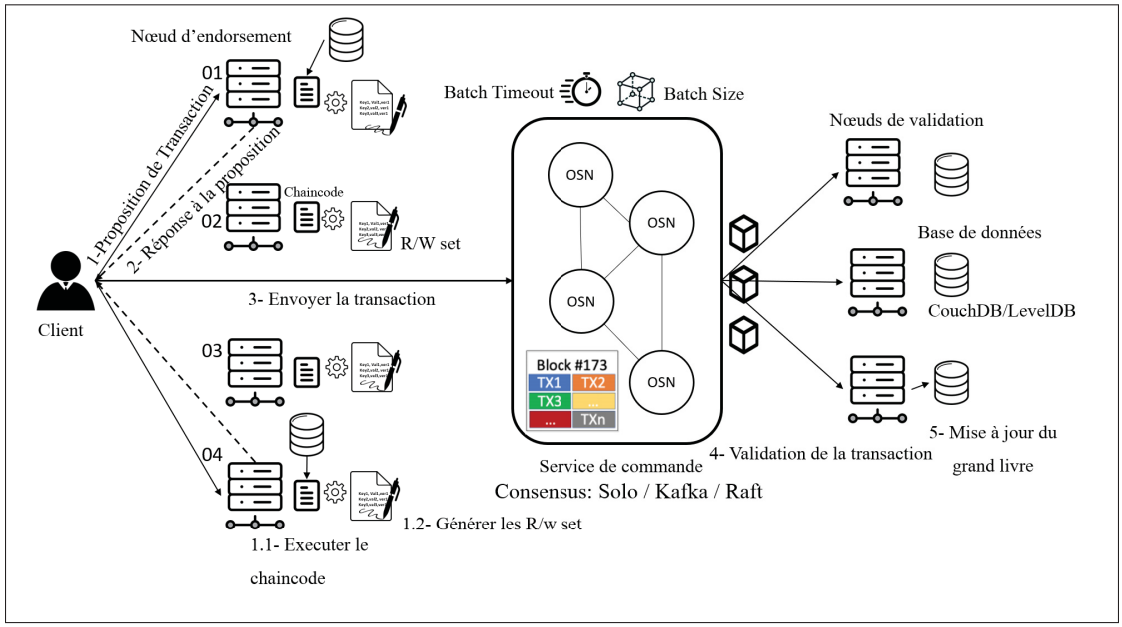


Figure 1.4 Scénario d'une transaction

1. **Étape 1** : un client qui souhaite effectuer une transaction envoie une proposition de transaction contenant les arguments et la fonction de chaincode qu'il souhaite invoquer aux pairs d'endossement conformément aux politiques d'endossement du chaincode (voir la section suivante 1.2.2).
2. **Étape 2** : un endosseur reçoit la proposition de transaction, exécute le chaincode et génère les ensembles de lecture/écriture qui sont les ensembles contenant les valeurs et les versions des clés lues ou écrites lors de l'exécution de la transaction, puis le pair crée la réponse à la proposition, la signe et l'envoie au client.
3. **Étape 3** : une fois que le client a collecté le nombre d'endossements requis, tel que défini dans les politiques d'endossement, il envoie la transaction aux services de commande contenant les détails de la proposition initiale ainsi que tous les endossements des pairs et les ensembles de lecture/écriture. Les services de commande ordonnent les transactions reçues des clients dans un bloc en tenant compte du BatchSize et BatchTimeout, puis le bloc est créé et diffusé aux noeuds de validation.
4. **Étape 4** : à la réception du bloc, le pair de validation parcourt toutes les transactions du bloc pour effectuer la validation syntaxique de chaque transaction, la validation VSCC et la validation MVCC (voir la section suivante 1.3).
5. **Étape 5** : si une transaction est marquée comme valide après avoir passé les trois vérifications, elle sera ajoutée au nouveau bloc et le grand livre est mis à jour en appliquant les ensembles d'écriture de transaction, sinon les transactions invalides seront rejetées et les clients devront les soumettre à nouveau.

1.3 Contrôle de la concurrence de multiversions

HLF utilise un système de contrôle de concurrence de multiversions pour assurer la cohérence du grand livre. Ce mécanisme valide que les versions des clés lues au moment de l'endossement de la transaction sont toujours les mêmes lors de la phase de validation (Larson *et al.*, 2012), ce processus garantit qu'il n'y a pas de lectures d'anciennes valeurs qui ont été modifiées par une autre transaction concurrente. Pendant le laps de temps entre l'endossement et la phase

de validation d'une TX2, si une autre transaction TX1 a mis à jour des clés répertoriées dans l'ensemble de lecture de la transaction TX2, cette dernière échouera à la validation MVCC.

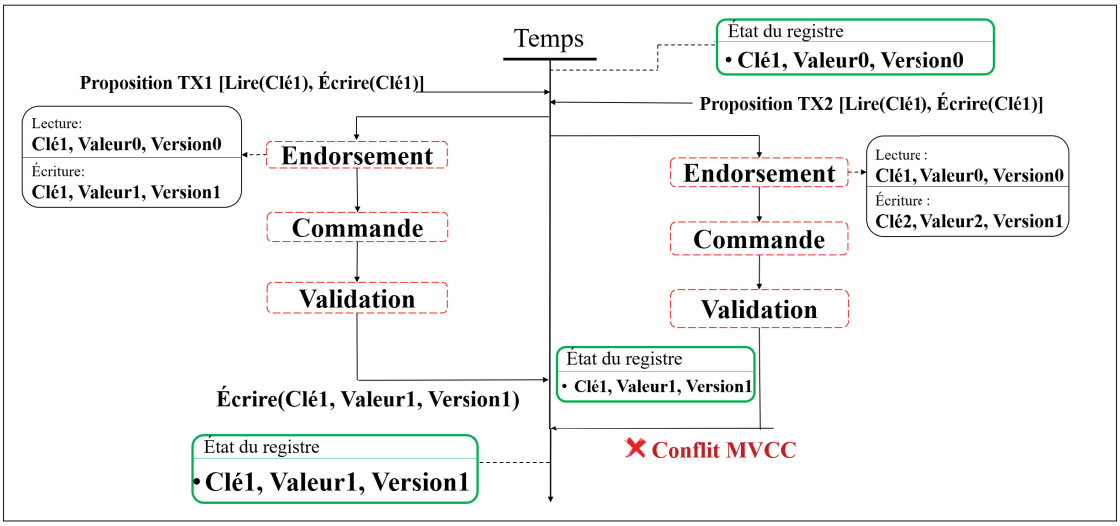


Figure 1.5 Illustration du conflit MVCC

Le conflit de lecture-écriture d'accès concurrentiel de multiversions est un problème qui se produit lorsque deux clients tentent de mettre à jour et de lire la même clé en même temps. La figure 1.5 montre un exemple de ce conflit MVCC. Supposons que *utilisateur1* soumet *TX1*, et en même temps *utilisateur2* soumet *TX2*, les deux transactions liront et mettront à jour la même valeur *Valeur1* de *Clé1*. Après avoir être endossée et ordonnancée simultanément, *TX1* sera validé et son ensemble d'écriture sera appliqué au grand livre entraînant la modification de la valeur et de la version associée à *Clé1* en *valeur1* et *version1*. Cependant, lorsque *TX2* subit la validation MVCC, elle échoue car la version de *Clé1* dans l'ensemble de lecture de la transaction n'est pas la même dans le grand livre. Par conséquent, la validation MVCC détecte l'incohérence entre le grand livre et le résultat de l'endossement et retourne une erreur MVCC.

1.4 Conclusion

Dans ce chapitre, nous avons introduit les concepts de base utilisés dans le cadre de ce mémoire tels que la technologie de chaînes de blocs ainsi qu'une comparaison entre les différents types de systèmes chaînes de blocs. Ensuite nous avons présenté deux théorèmes CAP et DCS qui

sont importants à comprendre pour analyser les systèmes distribués et les chaînes de blocs. Bien évidemment, nous avons présenté un survol de HLF pour présenter ses différentes composantes ainsi que le flux de la transaction. Enfin, nous avons expliqué le fonctionnement du mécanisme de détection de conflit de versions de HLF (MVCC).

CHAPITRE 2

REVUE DE LITTÉRATURE

HLF est un framework relativement nouveau qui connaît déjà des améliorations architecturales majeures. La majorité des travaux connexes visent à améliorer le débit et minimiser la latence du réseau, mais il existe un manque de solutions efficaces pour traiter le problème MVCC et dans la plupart des cas, le facteur de conflit de transaction n'est pas pris en compte lors de l'évaluation des performances. Dans cette section, nous passons en revue les recherches récentes sur les techniques permettant d'améliorer les performances des systèmes de chaîne de blocs publics en générale ainsi que les réseaux HLF. Nous passerons en revue des travaux sur les réseaux HLF selon quatre catégories : les travaux qui font une étude de performances,les travaux qui optimisent la phase d'endossement, les travaux qui améliorent la phase de commande et les travaux qui améliorent la phase de validation.

2.1 Optimisation des systèmes de chaîne de blocs publics

Afin de proposer une meilleure approche d'optimisation de notre système de chaine de blocs privée, nous avons aussi jeté un coup d'œil sur les propositions d'optimisation des performances des systèmes de chaines de blocs publics qui pourront nous servir dans la conception de notre solution. En effet, on a identifié des solutions d'optimisation des efforts du minage afin de résoudre le problème d'hétérogénéité de puissance des différents nœuds et des algorithmes de consensus tel que le travail de (Li & Li, 2020) qui propose un modèle numérique MENM (Miner Efficiency Numerical Model) pour mesurer l'efficacité de calcul du mineur, et un mécanisme de réglage des performances DDCT (Dynamic Difficulty Calculation and Tuning) pour ajuster dynamiquement la difficulté d'un mineur dans un groupe basé sur la valeur MENM, pour améliorer l'efficacité du groupe de minage. Le mécanisme de réglage dynamique de la difficulté DDCT a également prouvé son efficacité dans le réglage des performances du groupe de minage utilisant le consensus de la preuve de travail, confirmant que l'état global de la puissance de calcul du groupe de minage n'est pas une simple agrégation de tous les appareils connectés. Il y

a aussi un travail de (Qin, Yuan & Wang, 2018) qui étudie le problème de sélection de groupes auquels sont confrontés les mineurs et le modélisons comme un problème de décision de risque puisque les différents mécanismes de récompense ont des risques différents en établissant un modèle de sélection de groupe basé sur le critère de vraisemblance maximale afin de maximiser le profit du mineur. Aussi (Han, Foutris & Kotselidis, 2019a) ont élaboré une analyse approfondie des performances et une caractérisation de l'algorithme de preuve de travail afin d'appliquer une série d'optimisations sur l'algorithme de Ethereum Ethash. Les protocoles de chaine de blocs dérivés de Bitcoin ont des limites d'évolutivité, de débit et la latence, ce qui empêche la réalisation de ce potentiel. (Eyal, Gencer, Sirer & Van Renesse, 2016) présentent Bitcoin-NG (Next Generation), un nouveau protocole de chaine de blocs conçu pour remédier au problème d'évolutivité. Bitcoin-NG est un protocole de chaine de blocs byzantin tolérant aux pannes, qui est robuste partage le même modèle de confiance comme Bitcoin. Un autre volet d'optimisation a été traité par les travaux de recherches sur les systèmes de chaines de blocs publics, qui consiste à optimiser la consommation de gaz représentant le cout de la transaction. Il est souvent dominé par les instructions qui accèdent au stockage, GASOL (Albert, Correas, Gordillo, Román-Díez & Rubio, 2020) utilise l'analyse de gaz pour détecter les des modèles de stockage sous-optimisés. Il inclut un module d'optimisation automatique d'une fonction sélectionnée.

2.2 Étude des performances de HLF

Avant d'entamer l'optimisation des performances du réseau HLF, il est indispensable d'identifier les goulots d'étranglement principaux qui dégradent les performances du système. Il est aussi utile d'analyser l'impact des paramètres du réseau et du flux des transactions sur les différentes métriques de performances. Les auteurs (Thakkar, Nathan & Viswanathan, 2018) ont élaboré ce type de travail. Ils ont commencé par une analyse de l'impact des différents paramètres de configuration tels que la taille du bloc, les politiques d'approbation, le nombre de canaux et les ressources allouées sur le débit et la latence des transactions afin de fournir des lignes directrices pour la configuration de réseau HLF. Cette analyse a permis aussi l'identification de trois goulots d'étranglement qui été la vérification de la politique d'approbation, la validation

séquentielle des transactions par bloc et la mise à jour du registre partagé. Aussi (Androulaki *et al.*, 2018) ont détaillé l'architecture et le fonctionnement de HLF ainsi qu'une analyse de l'impact de la variation de la taille du bloc, des ressources de calcule et du nombre de pairs sur le débit des transactions. Dans le même sens (Baliga *et al.*, 2018) ont étudié l'impact des politiques d'approbation, la taille des ensembles de lecture et écritures et la taille du payload de la transaction etc. Ils ont aussi conduit des tests de scalabilité de réseaux en essayant de varier le nombre de chaincode invoqués ainsi que la nombre de canaux utilisés. Vu que la communication entre les noeuds du réseau HLF se fait sur un réseau de communication public, le travail de (Nguyen, Jourjon, Potop-Butucaru & Thai, 2019) a étudié l'impact des délais de réseau sur les performances du système ainsi il a défini les limites de délais tolérées pour que le réseau reste fonctionnel.

2.3 Optimisation de la phase d'endossement

Plusieurs techniques d'optimisation sont proposées dans les travaux de recherche associés. En effet, (Kwon & Yu, 2019) améliorent le traitement des transactions de lecture en distinguant entre les transactions de lecture et d'écriture pendant le processus d'endossement. En conséquence, la latence d'approbation des transactions est réduite de 60% par rapport au réseau Fabric traditionnel.

2.4 Optimisation de la phase de commande

Pour la phase de commande, (István, Sorniotti & Vukolić, 2018) proposent la suppression de la notion de blocs au profit du traitement des transactions en streaming pour diminuer la quantité de données à traiter. En outre, les auteurs implémentent un consensus basé sur une carte FPGA (Field-Programmable Gate Array) pour le service de commande qui permet de réduire la latence de validation en dessous d'une milliseconde en réduisant la latence à moitié par rapport au service de commande basé sur le consensus Raft. Les auteurs de Fast Fabric (Gorenflo, Lee, Golab & Keshav, 2019b) ont modifié le service de commande pour qu'il opère seulement avec les identifiants des transactions en séparant l'en-tête du reste des données de la transaction pour

ordonner les transactions uniquement avec leurs identifiants. Ceci accélère le traitement des transactions dans les services de commande ce qui augmente le débit. (Sharma, Schuhknecht, Agrawal & Dittrich, 2019) proposent Fabric++ une solution qui vise à réduire le taux d'échec MVCC en réorganisant les transactions lors de la phase de commande en utilisant un graphe de conflit pour abandonner les transactions qui ne peuvent pas être sérialisées. (Ruan *et al.*, 2020) a conçu une extension optimisée de Fabric++ qui peut gérer à la fois les conflits MVCC interblocs et intrablocs.

2.5 Optimisation de la phase de validation

Le troisième groupe représente les optimisations de la phase de validation, plusieurs articles (Gorenflo, Golab & Keshav, 2019a) (Gorenflo *et al.*, 2019b) (István *et al.*, 2018) (Javaid, Hu & Brebner, 2019) (Thakkar *et al.*, 2018) proposent l'exécution parallèle du processus de validation (Vérification des syntaxes, vérification des politiques d'approbation, validation MVCC) pour accélérer la validation des blocs. Les auteurs de FabricCRDT (Nasirifard & Mayer, 2019) se concentrent sur la fusion automatique des transactions en conflit à l'aide des techniques CRDT afin de diminuer les collisions et améliorer le débit effectif sans rejeter la transaction à la phase de validation. (István *et al.*, 2018) introduisent un mécanisme de traitement par lots des écritures sur disque en utilisant un batcher local qui accumule les opérations de mise à jour du grand livre jusqu'à atteindre une taille optimale du lot ou un délai d'attente pour écrire dans la base de données. Ajouté aux écritures par lots (Thakkar *et al.*, 2018) propose Bulk Read lors de la validation MVCC pour effectuer un seul appel par bloc de l'API de la base de données. Enfin, (Meir, Barger, Manevich & Tock, 2019) a présenté une solution sans verrou pour l'isolation des transactions, cette approche permet la suppression du verrou partagé tout en assurant l'isolation des transactions.

2.6 Lacune des travaux existants

Tous ces travaux offrent des solutions diverses pouvant améliorer les performances de HLF. Cependant, on peut constater que la majorité des améliorations touchent les phases de commande

et de validation. En plus, il y a une pénurie de solutions qui permettent de résoudre les conflits MVCC et dans la majorité des cas ce facteur très important d'évaluation des performances des réseaux HLF n'est pas pris en compte lors de la présentation des résultats des solutions proposées. On a remarqué qu'aucun de ces travaux n'apporte une solution pour détecter le conflit MVCC d'une manière précoce lors de la phase d'endossement.

2.7 Conclusion

Dans ce chapitre, nous avons étudié les travaux connexes à notre sujet de recherche. Nous avons pu présenter les principaux travaux qui ont étudié et évalué le système HLF ainsi que les travaux qui ont proposé des solutions d'optimisation des performances de ce réseau touchant aux trois phases d'exécution de la transaction. Enfin nous avons analysé les principales lacunes des travaux existants ce qui nous a permis de définir notre problématique de recherche.

CHAPITRE 3

ARCHITECTURE ET SOLUTIONS PROPOSÉES

Dans ce chapitre, nous présentons les solutions qu'on a discutées pour faire la détection précoce des conflits MVCC. Ensuite, nous introduisons la solution qu'on a adoptée ainsi que les différentes structures de données utilisées pour faire trois implémentations différentes. À la fin on va parcourir les étapes qu'on a suivies pour mettre en place cette solution.

3.1 Solutions suggérées

Après avoir identifier notre problématique qui était la détection précoce des conflits MVCC, on a commencé à définir l'architecture de notre système qui permet de sauvegarder les clés en cours d'utilisation. On a discuté trois solutions possibles :

- **Solution 1** : la Figure 3.1 illustre la première solution qui a été proposée. Elle consiste à utiliser une base de données partagée entre les différents pairs où on sauvegarde les clés qui sont en cours d'écriture et à chaque validation de transaction on met à jours cette base de données. Cette solution parait efficace pour la détection des conflits, mais on a jugé que la lecture et l'écriture sur la base de données engendrent des délais qui vont ralentir le traitement des transactions.

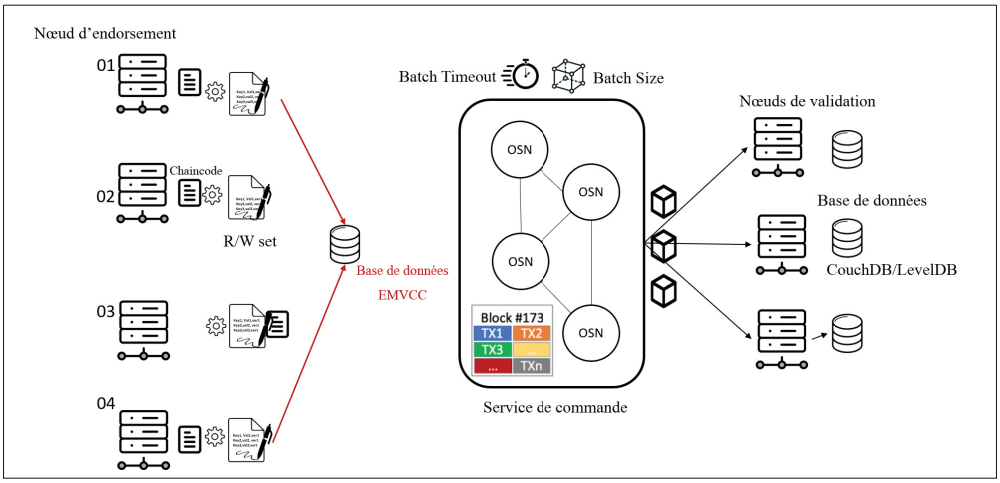


Figure 3.1 Solution avec une base de données EMVCC

- **Solution 2** : la Figure 3.2 illustre la deuxième solution qui consiste à implémenter un protocole de synchronisation entre les nœuds pour avoir une version unique de la liste des clés en cours de traitement. Cette solution est aussi efficace mais la synchronisation ajoute des délais supplémentaires qui augmentent la latence des transactions.

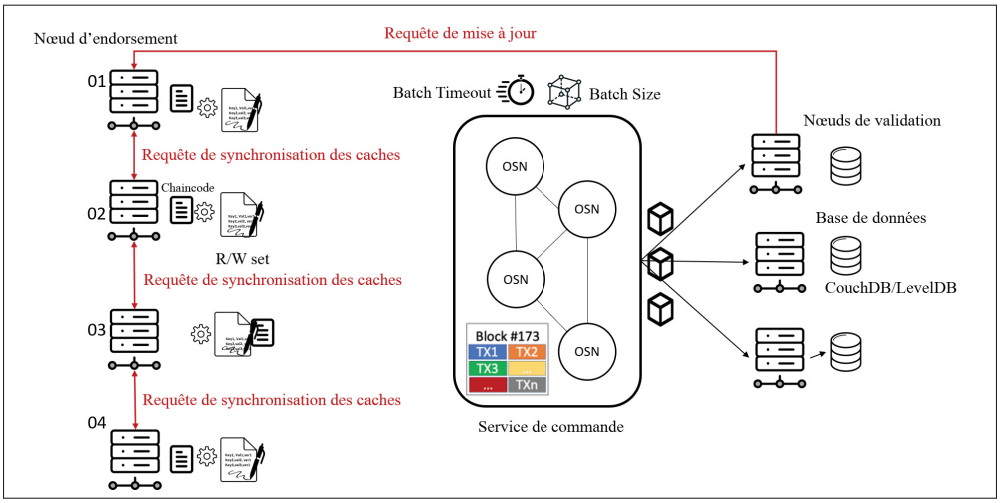


Figure 3.2 Solution avec un protocole de synchronisation

- **Solution 3** : la troisième solution est la celle que nous avons adopté et implémenté. Elle décrite en détail à la section suivante (Section 3.2). On a décidé de sacrifier la cohérence totale de la liste des clés en cours de traitement pour éviter d'ajouter de la latence aux transactions. Nous sauvegardons la liste des clés en cours de traitement dans la mémoire vive du nœud qui a endossé la transaction.

3.2 Architecture de la solution proposée

La présentation de notre solution de détection précoce de conflit MVCC se base essentiellement sur deux volets : en premier lieu la présentation des détails du fonctionnement de notre solution pour présenter en quoi diffère notre méthode par rapport à la version basic de HLF. En deuxième lieu, nous présentons les différentes structures de mise en cache que nous avons implémenté et comparé pour notre solution.

3.2.1 Présentation de la solution

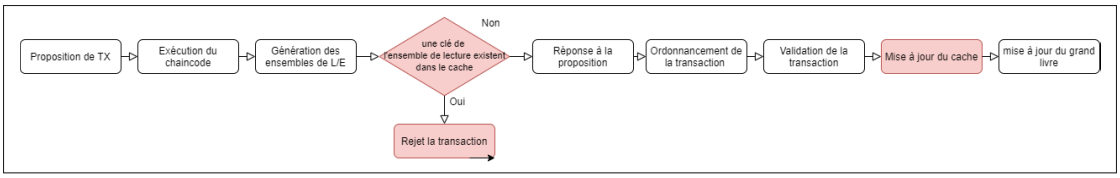


Figure 3.3 Architecture du mécanisme de détection précoce des conflits MVCC

Une transaction en conflit passe à travers tous le réseau pour finir à échouer à la phase de validation MVCC, ce fonctionnement cause la consommation des ressources pour le traitement de cette transaction sans aboutir à un résultat ce qui dégrade les performances du réseau. La solution que nous proposons, ajoute une couche de filtrage des transactions lors de la phase d'endossement pour abandonner les transactions en conflit au moment de l'endorsememt que nous appelons le contrôle de concurrence multiversion précoce (EMVCC). Comme décrit sur la figure 3.3, après avoir exécuté le chaincode et généré les ensembles de lecture/écriture, l'ensemble de lecture est comparé à la liste des transactions d'écriture en cours, stockée dans la mémoire vive du nœud endosseur. Si une transaction lit une clé stockée dans le cache, la transaction sera rejetée parce qu'une transaction qui fait une écriture sur la même clé est en cours. Une fois la transaction validée ou rejetée lors de la phase de validation, le cache est mis à jour en supprimant les clés de l'ensemble d'écriture de cette transaction. Cette couche ajoutée n'affecte pas la cohérence de HLF car toutes les transactions passeront la validation MVCC lors de la phase de validation.

La Figure 3.4 illustre un exemple du fonctionnement de la détection EMVCC. Nous supposons que nous avons un réseau mis en place composé de deux organisations, chaque organisation a deux pairs. La politique d'approbation du chaincode est `AND(Org1,Org2)`. Supposons que le client1 soumet TX1 qui écrit sur les clés clé1 et clé2. Cette transaction est approuvée par Pair0.ORG1 et Pair0.ORG2. Un exemple de transaction détectée avec succès est TX2 qui lit la clé2 et écrit sur la clé3. Si cette transaction est approuvée par Pair0.ORG1 ou Pair0.ORG2 elle sera abandonnée à la phase d'endossement car clé2 est dans le cache EMVCC. Pour TX3 qui

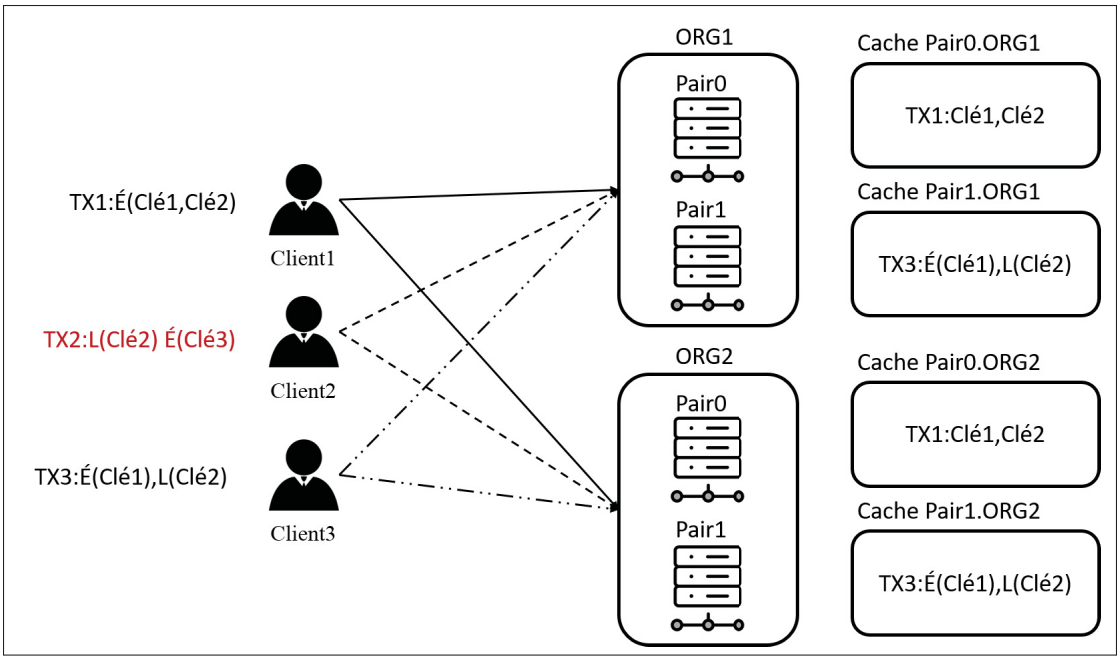


Figure 3.4 Scénarios de détection EMVCC

écrit sur clé1 et lit clé2, si elle est endossée par Pair1.ORG1 et Pair1.ORG2, le conflit ne sera pas détecté par le module de détection EMVCC car ces pairs n'ont pas de conflit avec cette clé dans leur cache EMVCC mais il échouera lors de la phase de validation MVCC.

Les algorithmes 3.1 et 3.2 représentent les pseudo-codes qui permettent d'identifier les principales modifications apportées par nos solutions respectivement à la phase d'endossement et de validation. Pour la phase d'endossement, une fois la proposition de la transaction est simulée on récupère les résultats de la simulation d'où on peut extraire les ensembles de lecture et d'écriture. Ensuite, on valide la transaction en faisant la recherche des clés au niveau de la l'ensemble de lecture dans le cache EMVCC, si la clé existe dans le cache la transaction sera rejetée sinon on ajoute l'ensemble d'écriture de la transaction dans le cache et procède à la formation de la réponse à la proposition. Quant à la phase de validation les modifications consistent principalement à faire la mise à jour du cache une fois la transaction est approuvée ou rejetée.

Algorithme 3.1 Pseudocode de l'endossement

```
Input : EMVCC_Validator : Structure de données qui stocke les clés

Lors de simulation de la proposition de la transaction
simResult := GetTxSimulationResults()
reads := simResult.GetReads() writes := simResult.GetWrites()
TxValidationCode := Validate_TX(EMVCCValidator, reads)
if TxValidationCode! = Valid then
    Reject_TX()
end if
else
    AddWsetToEMVCCValidator(writes, TxID)
end if
Function Validate_TX(EMVCCValidator, reads)
for Key in reads do
    if Key == EMVCCValidator.Key then
        return False
    end if
end for
```

Algorithme 3.2 Pseudocode de la validation

```
Input : EMVCC_Validator : Structure de données qui stock les clés

Lors de la validation de la transaction
if TxValidationCode! = Valid then
    Reject_TX()
end if
else
    ProcessTx()
end if
MVCCValidator.RemoveFromMVCCValidator(tx.id)
```

3.2.2 Choix de la structure de mise en cache

Dans le but d'implémenter une structure de mise en cache pour chaque nœud du réseau afin de sauvegarder les clés ayant des transactions en cours d'écriture, on a implémenté trois structures de données afin d'évaluer laquelle est la plus performante pour notre cas d'utilisation.

3.2.2.1 Mutex Lock

Pour cette implémentation, nous utilisons une structure de données basée sur des verrous (Dalessandro, Dice, Scott, Shavit & Spear, 2010) pour assurer la cohérence du cache en utilisant un système multithread vu que les endosseurs ainsi que les validateurs accèdent au cache pour ajouter ou supprimer des clés. Par exemple, lorsqu'un endosseur lit ou écrit sur le cache, il sera verrouillé et le validateur ne pourra pas mettre à jour cette entrée dans le cache. Le principal inconvénient de cette approche est la prolongation des délais d'attente aux opérations de lecture et d'écriture qui se trouvent obliger d'attendre la libération du verrou pour pouvoir s'exécuter.

3.2.2.2 Lock Free

C'est une alternative importante aux structures de données basées sur les verrous. Avec cette approche, plusieurs threads peuvent accéder au cache simultanément. Les opérations de lecture et d'écriture sont stockées dans une mémoire tampon puis une fois le seuil de cette mémoire est atteint, le lot est appliqué au cache ce qui garantit que la majorité des threads progressent à chaque étape (Valois). Cette technique est utile pour diminuer le nombre de fois qu'un verrou est appliqué ou libéré.

3.2.2.3 SyncMap

Avant la version 1.9 du langage Golang , la plupart du temps, les développeurs utilisaient une structure de données basée sur des verrous pour résoudre le problème de concurrence d'accès aux structures de données. Après la publication de cette version, la structure de données SyncMap a été implémentée dans le prologiciel Sync. C'est une structure optimisée et sécurisée qui utilise une map appelée *dirty map* pour écrire de nouvelles valeurs ou des mises à jour permettant à l'opération de lecture d'être effectuée sans verrou. Cependant, le verrouillage est indispensable si de nombreuses threads écrivent simultanément dans le *dirty map* ou lors de la mise à jour de la structure en lecture seule par le lot d'opérations enregistrer dans le *dirty map*.

3.3 Étape d'implémentation

Afin d'implémenter les trois solutions proposées, nous avons fait des modifications sur le code source de HLF à la version 2.3. Nos modifications ont touché principalement les fichiers reliés aux endosseur et au validateur. On a aussi crée un nouveau progiciel pour ajouter les fonctions qui permettent de gérer notre structure de données utilisée. Après avoir fait toutes les modifications nécessaires, nous avons élaboré les tests unitaires et fonctionnels qui nous permettent de nous assurer du fonctionnement correct de notre solution. Enfin on passe à la génération des images docker qui sont utiles pour mettre en place un réseau HLF qui serait l'environnement d'évaluation de nos solutions.

3.4 Conclusion

Dans ce chapitre, nous avons présenté un survol des différentes solutions que nous avons étudiées pour mettre en place un système de détection précoce de conflit MVCC. Ainsi nous avons détaillé l'architecture de la solution qu'on a adopté et les différentes structures de données utilisées pour le stockage des clés. Enfin, nous avons détaillé les étapes de l'implémentation de notre solution de détection précoce de conflit MVCC.

CHAPITRE 4

ANALYSE THÉORIQUE

Dans ce chapitre, nous allons introduire la formulation de notre problématique après avoir modélisé notre système. Ensuite, nous présentons une analyse théorique de notre solution de détection précoce des conflits MVCC qui va nous servir pour évaluer nos résultats pratiques.

4.1 Modélisation du système

Dans cette section, nous utiliserons le modèle suivant : étant donné un réseau HLF contenant N organisations $(Org_1, Org_2..., Org_n)$ où chaque organisation a M pairs $(Pair_1, Pair_2..., Pair_m)$ et M_i est le nombre de pairs pour l'organisation i. Le système utilise $Nb_{Tx/Block}$ comme nombre de transactions par bloc. Les services de commande du système peuvent utiliser n'importe quel protocole de consensus disponible dans HLF (par exemple, Raft, Kafka, Solo). Le chaincode exécuté par l'application génère $\%conflict$ comme taux de conflit représentant le pourcentage de transactions échouées en raison d'une erreur de conflit MVCC. Intuitivement parlant, un chaincode où les transactions lisent et écrivent sur quelques clés partagées aura un taux de conflit plus élevé que celui d'un chaincode où les transactions s'exécutent indépendamment sur des clés disjointes. Afin de stocker l'état de la chaîne, le système peut être implémenté à l'aide de l'une des bases de données disponibles compatibles avec HLF (CouchDB, LevelDB, etc.).

4.2 Formulation du problème

Le problème que nous résolvons est la détection précoce des conflits MVCC en filtrant les transactions lors de la phase d'endossement en fonction de l'historique des transactions approuvées sur chaque pair endosseur. Nous commençons par définir quelques termes de base que nous utiliserons plus tard :

- **Probabilité de non-détection** : c'est une métrique pour évaluer les performances de la solution EMVCC. Une solution EMVCC parfaite aurait une $\mathcal{P}(ND) = 0$. Cela représente la

probabilité que notre solution proposée ne détecte pas une transaction conflictuelle lors de la phase d'endossement, même si elle est en conflit avec une autre transaction, elle passera la validation EMVCC mais échouera lors de la phase de validation MVCC ce qui fait que notre solution ne touche pas à la cohérence du système.

- **Faux positifs** : nous appelons un faux positif le fait qu'une transaction soit déclarée invalide à l'aide de notre validation EMVCC, cependant, elle est valide car la transaction avec laquelle il y a des conflits échoue pour une raison quelconque après la phase d'endossement. Idéalement, les faux positifs devraient être égaux à zéro et on peut les calculer comme suit :

$$\begin{aligned} Tx\_invalide\_ND &= Nb_{Tx/Bloc} \times \%conflit \times \mathcal{P}(ND) \\ Tx\_invalide\_D &= Nb_{Tx/Bloc} \times \%conflit \times (1 - \mathcal{P}(ND)) \end{aligned} \tag{4.1}$$

$$Tx\_passant\_endossement = Tx\_valide + Tx\_invalide\_ND$$

$$FP = Tx\_passant\_endossement \times \frac{Tx\_invalide\_ND}{Tx\_invalide\_D} \tag{4.2}$$

$$FP = \frac{Nb_{Tx/Bloc} \times \%conflit^2 \times \mathcal{P}(ND) \times (1 - \mathcal{P}(ND))}{\%conflict \times \mathcal{P}(ND) - \%conflicts + 1} \tag{4.3}$$

Où $Nb_{Tx/Bloc}$ est le taux d'arrivée des transactions, $\mathcal{P}(ND)$ est la probabilité de non-détection et $\%conflict$ est le taux de conflit.

- **Faux négatifs** : c'est l'erreur inverse où une transaction est déclarée valide à l'étape de validation EMVCC alors qu'elle est réellement en conflit avec une autre transaction entraînant son échec. Idéalement, les faux négatifs devraient être égaux à zéro et on peut les calculer

comme suit :

$$FN = Nb\_Transaction\_en\_conflit \times \mathcal{P}(ND)$$
$$FN = Nb_{Tx/Bloc} \times \%conflit \times \mathcal{P}(ND) \tag{4.4}$$

- **Goodput** (Yu, Xue, Kilari, Yang & Tang, 2018) : c'est le débit effectif qui représente le taux de transactions réussies que le réseau peut traiter étant donné un sous-ensemble de paramètres. Idéalement, le Goodput devrait être égal au débit. Nous pouvons le calculer comme suit :

$$Goodput = Poucentage\_Tx\_Réussi \times Débit$$
$$Goodput = \frac{Nb\_transaction\_valide}{Nb\_transactions\_total} \times Débit \tag{4.5}$$

4.3 Analyse théorique de la solution

Nous présentons un aperçu théorique de notre solution proposée, en utilisant notre implémentation chaque nœud du réseau aura une version de cache qui dépend des transactions endossées. Cette incohérence et ce manque de synchronisation entre les nœuds impactent la décision prise concernant la validité de la transaction à la phase d'endossement. Par conséquent, les principaux facteurs pouvant avoir un impact sur les performances de notre solution sont les politiques d'endossement utilisées et la topologie du réseau (nombre d'organisations et nombre de pairs par organisation). Nous analysons l'impact des politiques d'endossement AND et OR. Ces résultats théoriques seront comparés aux résultats réels dans la section 5 pour évaluer l'exactitude de notre implémentation. Nous avons également fait une analyse théorique de la politique de Out-of, mais cela sort du cadre de ce travail. Nous devons noter que nous supposons que le choix des pairs d'approbation est fait de manière aléatoire par le client pour satisfaire la politique d'approbation lors de l'envoi de la proposition de transaction (Manevich *et al.*, 2018).

4.3.1 Analyse théorique de la politique d'approbation AND

En modélisant notre système, nous pouvons calculer la probabilité qu'une transaction ne soit pas détectée par le filtrage EMVCC pour la politique d'approbation AND comme suit : supposons qu'on a deux transactions (*TX1* et *TX2*), le client soumet *TX1* qui est endossée par un pair de chaque organisations sur le réseau. Ensuite une deuxième transaction *TX2* ayant un conflit MVCC avec *TX1* à été soumise. La probabilité de non détection est la probabilité que la de transaction *TX2* ne soit pas endossée par un pair qui a endossé la transaction *TX1*, c'est à dire la probabilité de non détection est le produit des probabilités de non détection conditionnelles de chaque organisations. La probabilité de non détection sachant qu'un pair appartient à une organisation est le nombre de pairs qui n'ont pas endossé *TX1* qui est $(M_i - 1)$ divisé par le nombre total des pairs de cette organisation (M_i).

$$\mathcal{P}(ND) = \prod_{i=1}^{N} P(ND \mid Org_i)$$
$$\mathcal{P}(ND) = \prod_{n=1}^{N} \frac{(M_i - 1)}{M_i} \tag{4.6}$$

La figure 4.1 trace l'évolution théorique de la probabilité de non-détection de la solution EMVCC pour diverses organisations et nombres de pairs utilisant la politique d'approbation AND. Nous observons que la probabilité de non-détection diminue lorsque le nombre d'organisations augmente. Cependant, elle augmente lorsque le nombre de pairs par organisation augmente. Plus le réseau contient d'organisation, plus la solution EMVCC est performante en utilisant la politique AND.

4.3.2 Analyse théorique de la politique d'approbation OR

De même, on peut calculer la probabilité de non détection de transaction en conflit pour la politique d'approbation OR comme suit : si les organisations ont le même nombre de pairs, la probabilité de non détection est le résultat de la substitution de un moins la probabilité de détection. Pour M pair par organisation, la probabilité de détection est de $\frac{1}{NM}$.

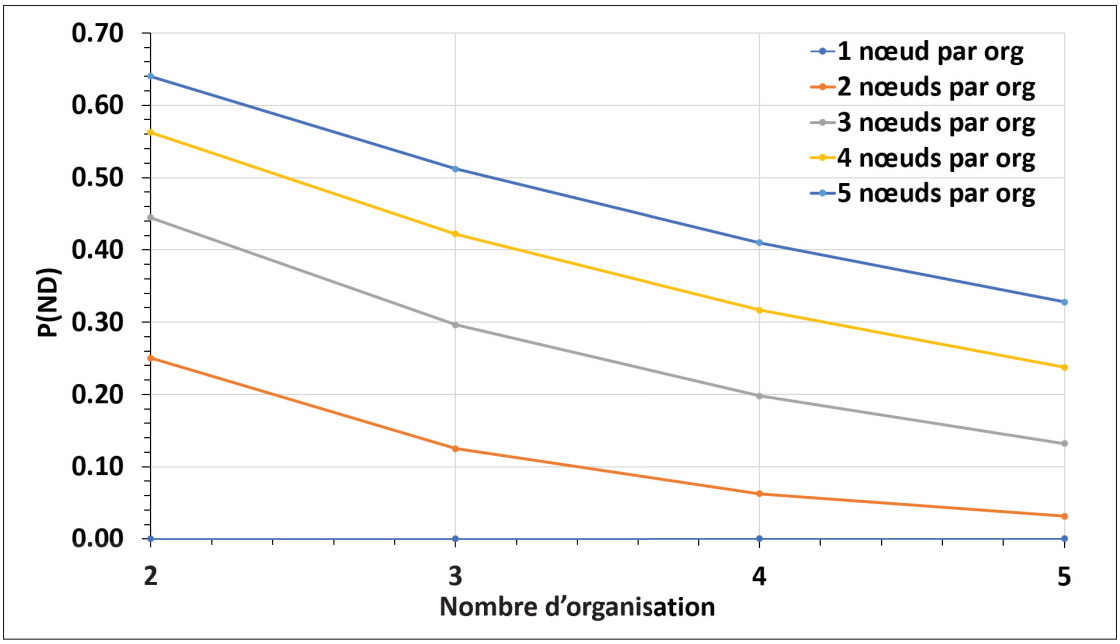


Figure 4.1 Probabilité de non détection en utilisant la politique d'approbation AND

$$\begin{aligned} \mathcal{P}(ND) &= 1 - \mathcal{P}(D) \\ \mathcal{P}(ND) &= 1 - \frac{1}{NM} \\ \mathcal{P}(ND) &= \frac{NM - 1}{NM} \end{aligned} \tag{4.7}$$

Sinon si les organisations ont des nombres de pairs différents, on calcule la probabilité de non détection de la manière suivante : c'est la somme des produits de probabilité que la transaction sera endossée par une organisation et la probabilité conditionnelle de non détection par organisation. La probabilité d'avoir un endossement d'un pair particulier est de $\frac{1}{N}$ et la probabilité conditionnelle de non détection par organisation représente la probabilité de recevoir un endossement d'un pair qui n'a pas endossé la première transaction qui est équivaut à $\frac{1}{M_i}$.

$$\mathcal{P}(ND) = \sum_{i=1}^{N} \mathcal{P}(Org_i) \times P(ND \mid Org_i)$$

$$\mathcal{P}(ND) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{C_{M_i}^{M_i-1}}$$

$$\mathcal{P}(ND) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{M_i}$$

Si on résume, on obtient la formule de calcule de la probabilité de non détection en utilisant la politique d'approbation OR :

$$\mathcal{P}(ND) = \begin{cases} \frac{NM-1}{NM} & \text{si } M_i{=}M_{i+1} \\ \frac{1}{N} \sum_{i=1}^{N} \frac{1}{M_i} & \text{sinon} \end{cases} \tag{4.8}$$

La figure 4.2 illustre l'évolution théorique de la probabilité de non-détection pour différents nombres d'organisations et nombre de pairs en utilisant la politique d'approbation OR. Nous constatons que la probabilité de non-détection augmente lorsque le nombre d'organisations ou le nombre de pairs augmente en raison du manque de synchronisation entre les caches des nœuds. Plus le nombre d'organisations augmente, moins la solution est performante.

4.4 Conclusion

Dans ce chapitre, nous avons fourni la formulation du problème, ainsi nous avons défini les termes qu'on va utiliser pour évaluer nos solutions et les principales formules mathématiques qui permettent de calculer ses métriques. Enfin nous avons réalisé une analyse théorique de l'évolution de la probabilité de non détection pour avoir un aperçu sur l'utilité de la solution et de comparer ces résultats théoriques aux résultats pratiques dans le but de valider nos résultats.

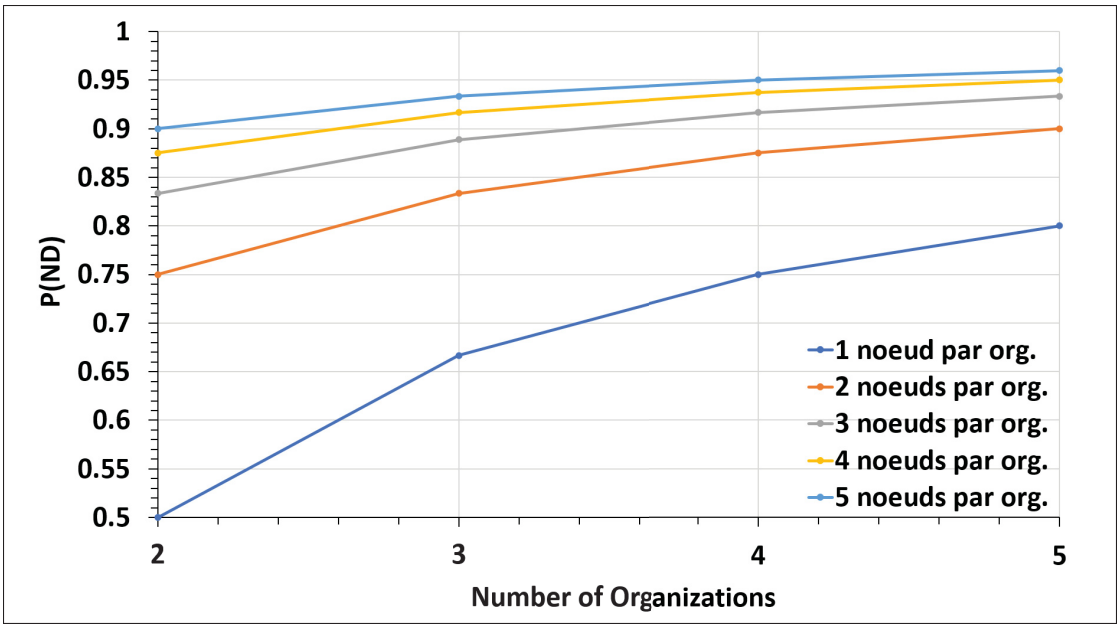


Figure 4.2 Probabilité de non détection en utilisant la politique d'approbation OR

CHAPITRE 5

ÉVALUATION ET DISCUSSION DES RÉSULTATS

Dans le chapitre précédent, nous avons présenté les principales modifications apportées à l'architecture basique de HLF. Dans ce chapitre nous allons évaluer l'efficacité de nos solutions par rapport à HLF. Nous commencerons notre évaluation par une comparaison de nos solutions proposées avec la référence HLF. Ensuite, nous faisant une analyse de sensibilité pour étudier l'influence de certains paramètres du système et des caractéristiques de la charge de travail sur les performances du système. Le tableau 5.1 montre les paramètres utilisés par défaut pour nos expériences.

Tableau 5.1 Paramètres d'évaluation par défaut du réseau HLF

Paramètres	**Values**
Taille du bloc	500 TX
Temps du bloc	2s
Nombre d'organisation	2
Nombre de pairs par organisation	2
Consensus	Raft (3 OSN)
Pourcentage de conflit	40%
politique d'approbation	AND(Org1,Org2)

5.1 Environnement de travail

Décrivons la configuration avant de commencer l'interprétation des résultats expérimentaux. Notre système d'évaluation se compose de six machines virtuelles E2 hébergées sur Google Cloud Platform telles qu'illustrées par la Figure 5.2. Deux machines servent comme deux pairs appartenant à deux organisations différentes et trois machines comme des noeuds du service de commande avec un algorithme de consensus Raft. La sixième machine virtuelle joue le rôle des clients en exécutant Hyperledger Caliper (Wickboldt, 2019) qui est un outil d'évaluation des réseaux HLF permettant de générer des transactions et de calculer des métriques de performances telles que le débit et la latence. Chaque machine virtuelle possède 8 vCPU et 32 Go de RAM.

Toutes les machines virtuelles exécutent Ubuntu 16.04 en tant que système d'exploitation. Les pairs sont configurés pour utiliser CouchDB comme base de données. Pour le chaincode, nous utilisons le chaincode Fabcar (Appendix II) qui nous permet d'enregistrer une voiture sur la blockchain et de modifier son propriétaire.

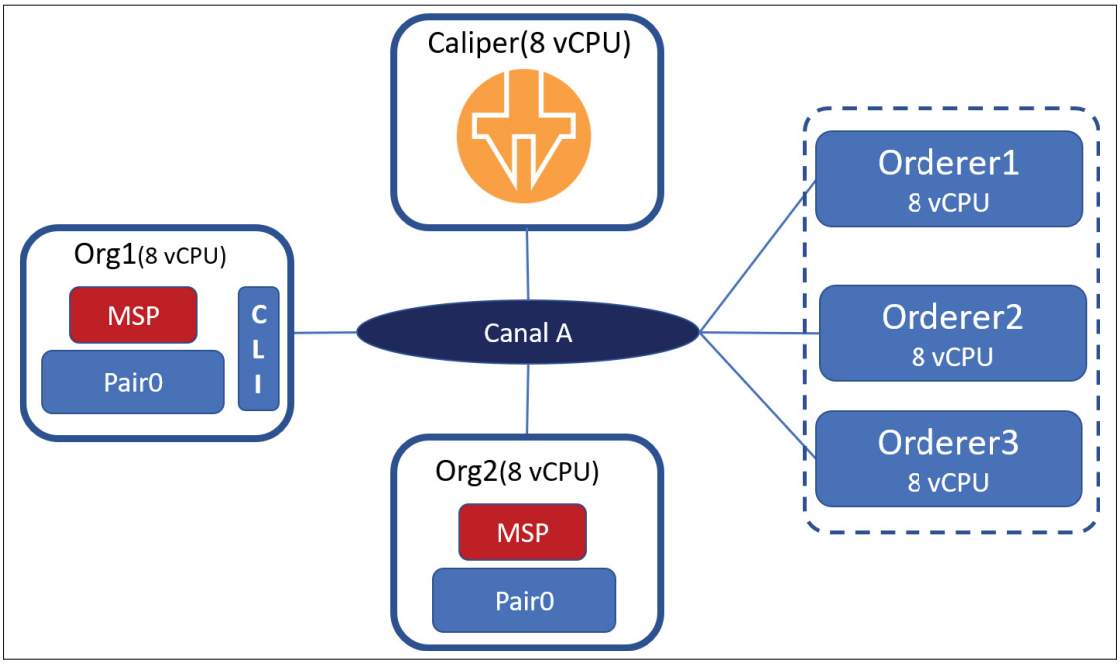


Figure 5.1 Réseau d'évaluation

5.2 Métriques de performance

Afin de faire l'évaluation, nous avons défini les métriques de performance qu'on va utiliser pour comparer les différentes implémentations :

1. **Goodput** : c'est le débit effectif des transactions validées (Section 4.2).
2. **Latence** : c'est le temps entre la soumission de la transaction par le client et son rejet ou application au registre partagé.
3. **Taux EMVVC vs MVCC** : c'est le pourcentage des transactions rejetées en raison d'une erreur EMVCC par rapport au pourcentage de transactions rejetées en raison d'une erreur MVCC.

4. **TD EMVCC et TD MVCC** : c'est le temps entre la soumission de la transaction par le client et son rejet par une erreur MVCC ou EMVCC.

5.3 Comparaison avec la version basique de HLF

Dans cette section, nous comparons nos trois solutions proposées avec la version 2.3 d'HLF. Nous prenons la moyenne du débit et de la latence des transactions sur dix exécutions.

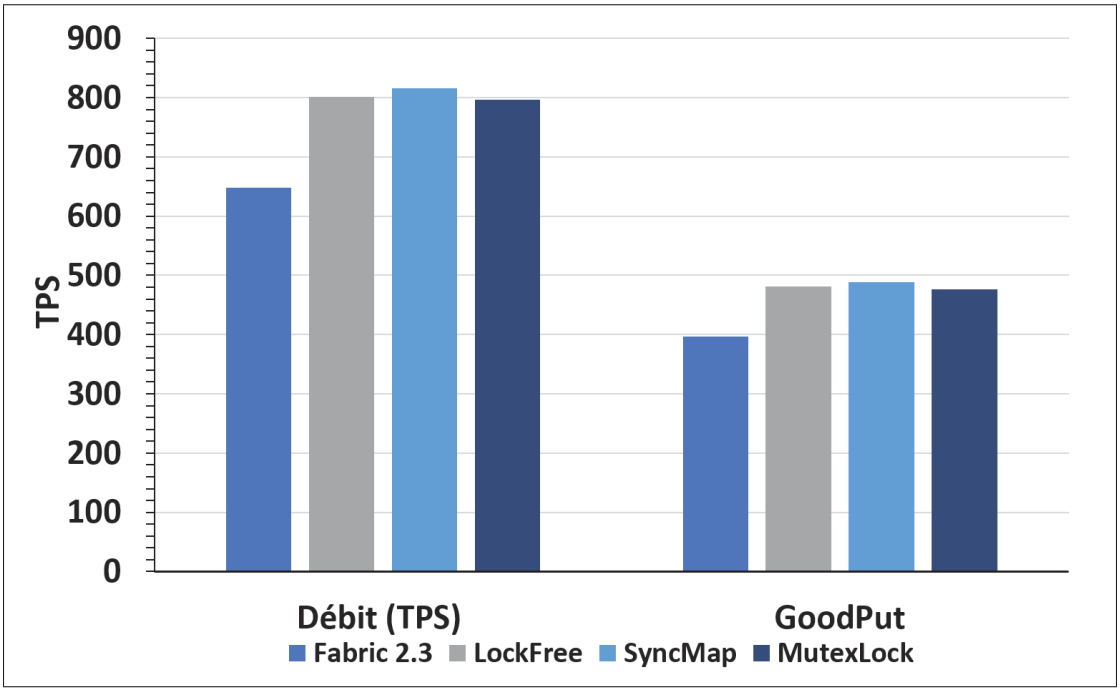


Figure 5.2 Comparaison du débit avec la référence de HLF

La figure 5.2 représente l'évolution du débit moyen et du Goodput pour les trois implémentations ainsi que la référence de HLF. Les trois solutions proposées offrent des performances plus élevées que la référence. La meilleure solution est SynMap avec 23,2% d'amélioration du Goodput par rapport au HLF. Pour Mutex Lock et LockFree, le pourcentage d'amélioration est respectivement de 20,4% et 21,2%. La figure 5.3 trace la latence moyenne des trois implémentations et la référence de HLF. Aussi, les trois solutions réduisent la latence, pour SyncMap et LockFree la latence est réduite de 80% et pour MutexLock de 69%. Ceci est dû à la détection précoce de la

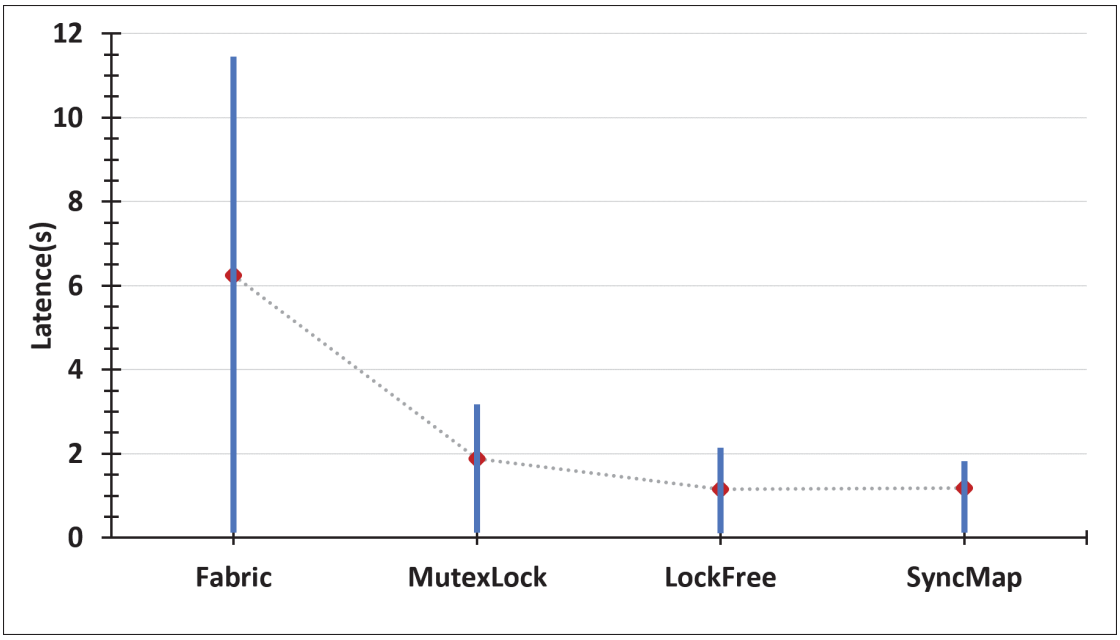


Figure 5.3 Comparaison de la latence avec la référence de HLF

transaction en conflit lors de l'endossement au lieu d'aller jusqu'à la phase de validation pour être rejetée. Nous pouvons confirmer que SyncMap est la meilleure structure de données pour notre implémentation de cache EMVCC puisqu'elle est la meilleure structure de données pour des opérations de lecture lorsque le pair a plusieurs vCPU.

5.4 Analyse de sensitivité

Dans cette section, nous analysons l'impact de divers paramètres de réseau et de charge de transaction sur les métriques de performances telles que le taux de conflit, les politiques d'approbation et les ressources de calcul par pair, etc.

5.4.1 Impact du taux de conflit

Le taux de conflit est le paramètre le plus important qui peut nous montrer l'utilité de notre solution proposée. Dans la figure 5.4, nous traçons le Goodput pour la référence et nos différentes implémentations pour des différentes valeurs de taux de conflit. Comme prévu, avec une

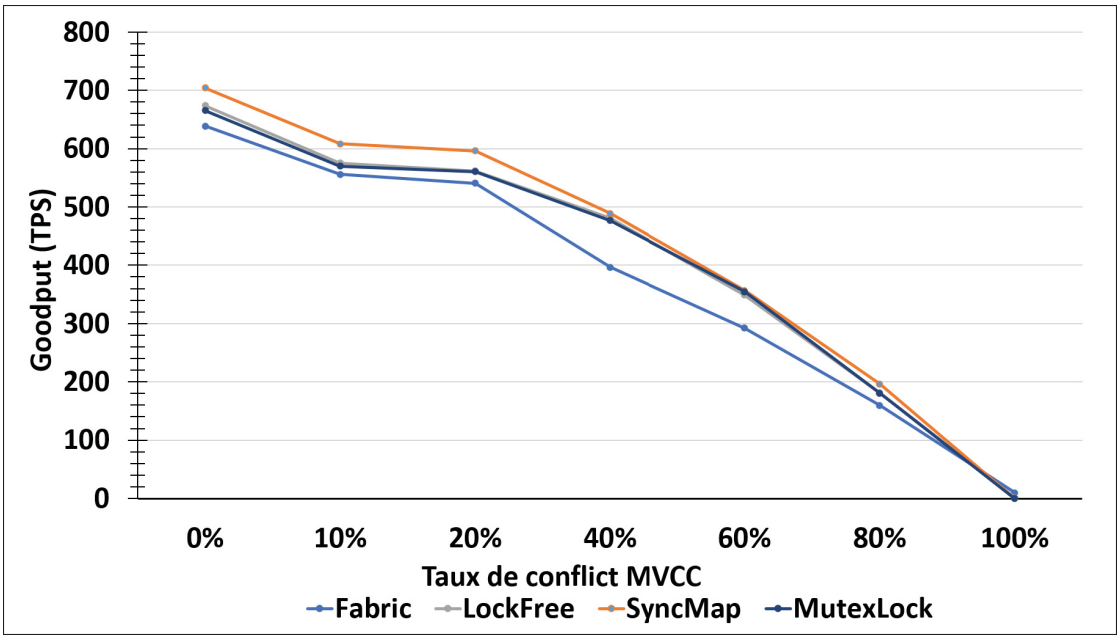


Figure 5.4 Taux de conflit vs Goodput

augmentation du taux de conflit, le Goodput diminue, car nous augmentons le nombre de transactions échouées jusqu'à atteindre 100% de conflit où le Goodput tend à être nul. De plus, nous pouvons voir que les trois solutions proposées fonctionnent mieux que HLF pour différentes valeurs du taux de conflit. SyncMap est la meilleure solution, elle améliore le Goodput de 10% lorsque le conflit est d'environ 20% et son impact est plus important en dépassant 20% de conflit pour atteindre une amélioration de 23% à un taux de conflit de 40% . Les performances de cette solution sont de plus en plus meilleures que la référence avec l'augmentation du taux du conflit de l'application.

5.4.2 Impact des ressources de calcule par pair

Dans cette partie, nous analysons l'impact des ressources de calcule par pair en variant le nombre de vCPU de 2vCPUs à 8 vCPUs. La figure 5.6 trace le débit moyen et le Goodput alors que la figure 5.9 présente la latence moyenne pour les trois implémentations et la référence de HLF en utilisant divers nombres de processeurs virtuels par pair. On constate qu'avec une augmentation du nombre de vCPU par pair, le Goodput augmente et la latence diminue. Nos trois solutions sont

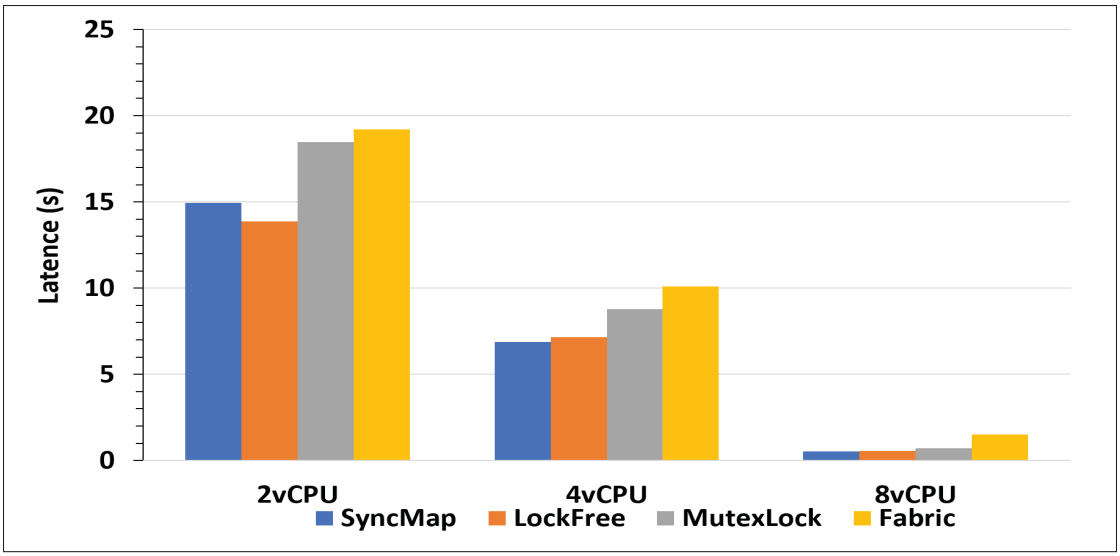


Figure 5.5 vCPUs vs Latence

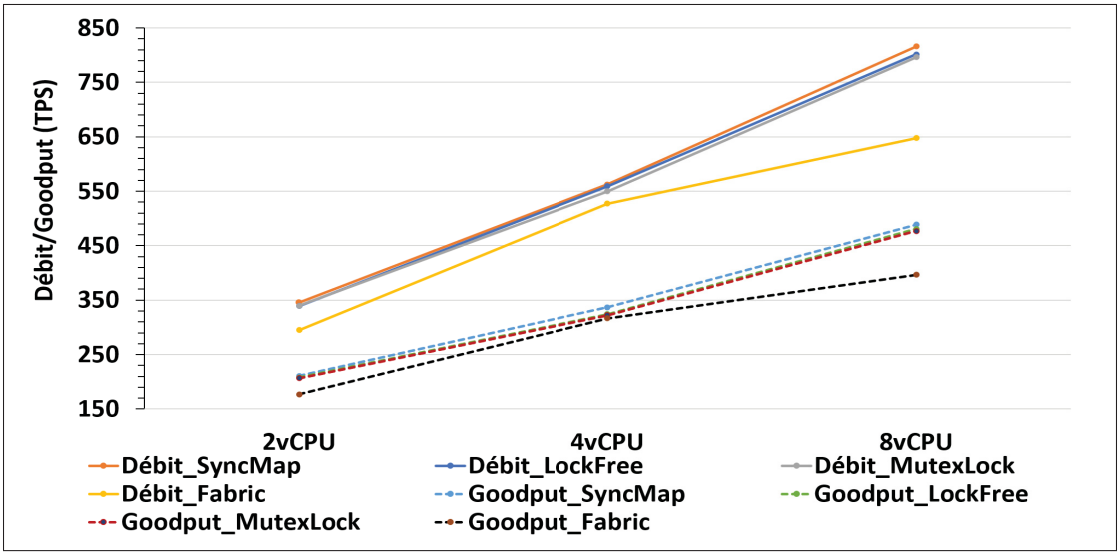


Figure 5.6 vCPUs vs Goodput

meilleures que la référence pour les différents nombres de vCPUs. Cependant, nous observons qu'avec 8vCPU une amélioration significative de 23% en termes de Goodput et de 65% en termes de latence entre la solution SyncMap et HLF, ce qui confirme que nos solutions sont plus performantes lorsque les pairs ont plus de puissance de calcul.

5.4.3 Impact de la taille du bloc

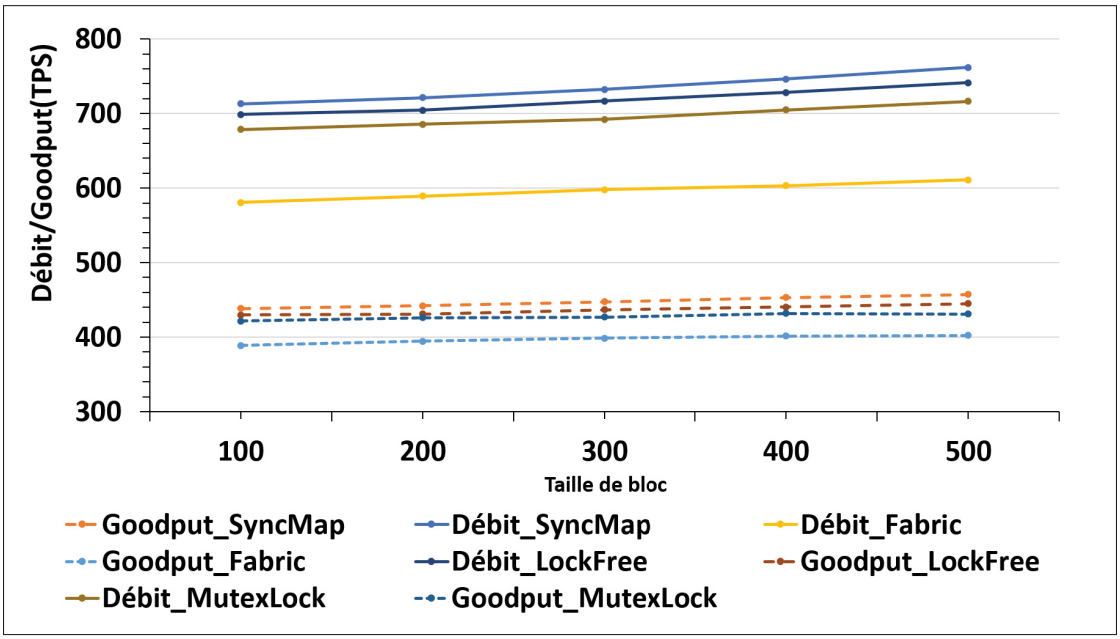


Figure 5.7 Taille du bloc vs Goodput

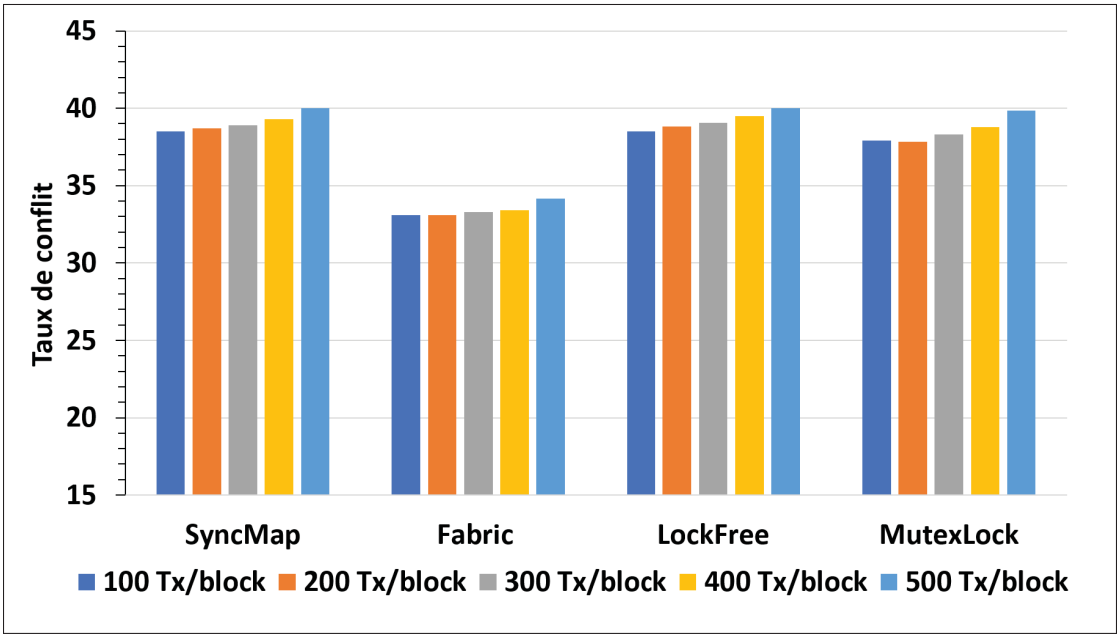


Figure 5.8 Taille du bloc vs Taux de conflit

Nous analysons l'impact de la taille du bloc en faisant varier ce paramètre de 100 transactions par bloc à 500 transactions par bloc. La figure 5.7 trace le débit moyen et le Goodput pour HLF et nos trois implémentations sur différentes tailles de blocs. Comme nous pouvons le voir, l'augmentation de la taille des blocs augmente le débit et le Goodput pour HLF ainsi que nos solutions, car l'utilisation de blocs plus grands entraînera moins de surcharge de communications. La méthode SyncMap offre une amélioration moyenne de 12% par rapport à HLF sur différentes tailles de blocs. La figure 5.8 représente le taux de conflit pour HLF et nos trois méthodes pour des différentes tailles de blocs. On peut observer qu'avec une augmentation de la taille du bloc le taux de conflit augmente, ceci est dû à l'augmentation du nombre de conflits intrabloc à cause du nombre élevé de transactions. On peut aussi observer que le taux de conflit pour HLF est inférieur à notre implémentation proposée, c'est parce qu'il y a quelques faux positifs et négatifs causés par le détecteur EMVCC.

5.4.4 Impact des politiques d'approbation et de la topologie du réseau

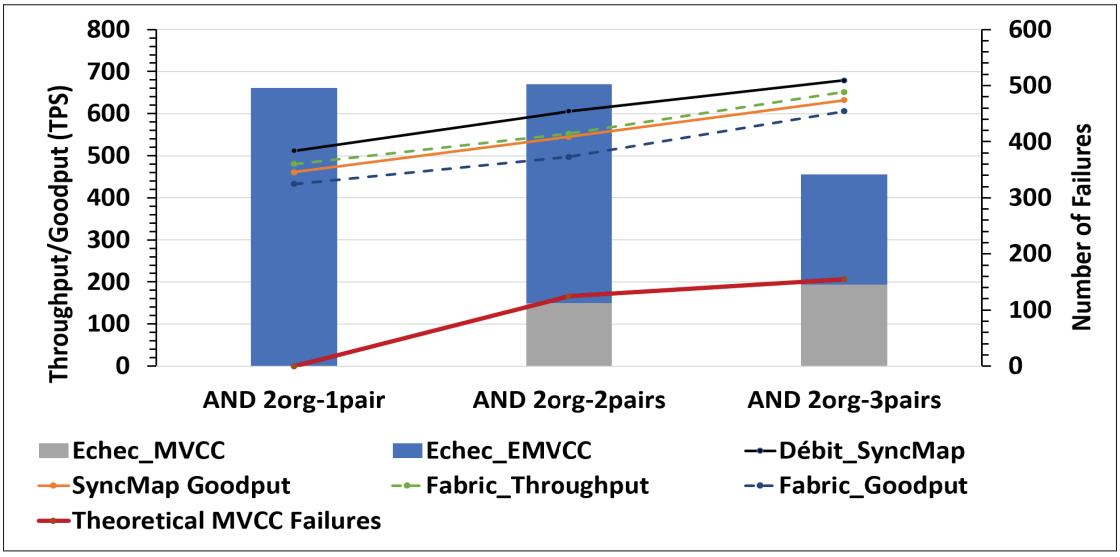


Figure 5.9 Impact de la politique d'approbation AND

Pour cette expérience, nous utilisons différentes topologies de réseau en faisant varier le nombre de pairs par organisation d'un pair à trois pairs. Les figures 5.9 et 5.10 tracent le débit moyen et le Goodput, ainsi que le pourcentage d'échecs de transaction causé par la validation EMVCC et

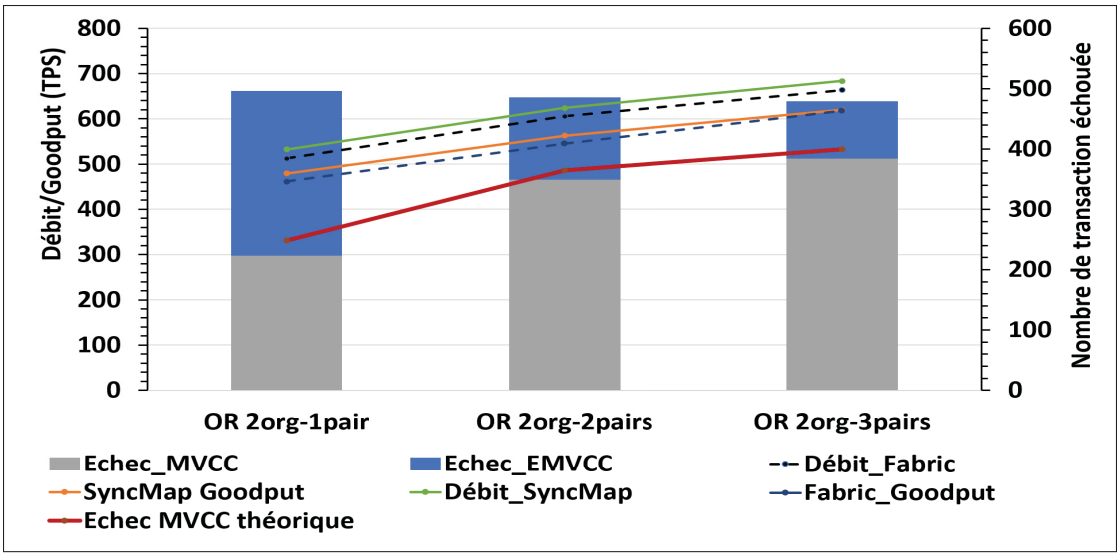


Figure 5.10 Impact de la politique d'approbation OR

MVCC pour différentes topologies de réseau à l'aide des politiques d'approbation `AND(Org1, Org2)` et `OU(Org1, Org2)` pour les solutions HLF et SyncMap. Nous observons que le type de politique d'approbation impacte les performances du réseau. Comme le montrent les deux figures 5.9 et 5.10, le débit de la politique d'approbation OR est supérieur au débit utilisant la politique AND, car en utilisant OR, la transaction nécessite moins d'endossement par les pairs que le AND. Pour le pourcentage de non détection, ces résultats expérimentaux confirment les résultats théoriques présentés dans la section 4.3. Par exemple, avec la politique AND et en utilisant un réseau avec deux organisations chacune à deux pairs, nous avons 22% de transactions conflictuelles qui n'ont pas été détectées par le module EMVCC et ont été rejetées à la validation MVCC (faux négatif). Nous pouvons également voir que l'augmentation du nombre de pairs augmente le débit et le Goodput pour les deux politique d'approbation, car les organisations disposent de plus de ressources pour traiter les transactions. L'utilisation de la politique d'approbation AND assure une amélioration significative par rapport à la référence de HLF, cependant, en utilisant la politique d'approbation OR, le nombre de faux positifs augmente, ce qui impact les performances de la solution. Nous pouvons conclure qu'avec une augmentation du nombre d'endosseurs, notre solution devient plus efficace.

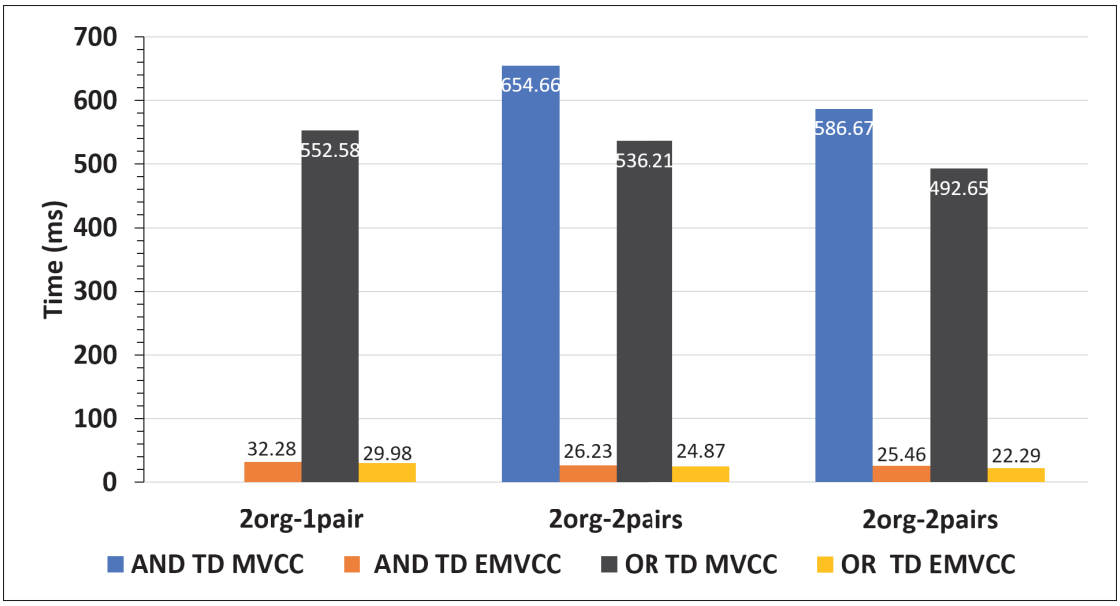


Figure 5.11 Temps de détection EMVCC/MVCC

La figure 5.11 trace le temps moyen pour détecter un conflit EMVCC ou MVCC pour les différentes topologies de réseau à l'aide des politiques d'approbation OR et AND. Nous pouvons constater une grande différence temporelle entre le rejet d'une transaction en conflit lors de la phase EMVCC et son traitement jusqu'à atteindre la phase de validation MVCC pour être abandonnée. Ce filtrage précoce réduit considérablement la latence des transactions. Nous notons que lorsque le nombre de pairs par organisation augmente, le temps de détection du MVCC diminue en raison de la disponibilité des ressources de calcul.

5.4.5 Impact de l'implémentation du chaincode

Afin de simuler le comportement du chaincode, nous utilisons la distribution Zipf (Eliazar, 2016), qui nous permet de choisir les clés utilisées pour simuler les transactions en faisant varier le paramètre s de la distribution de 0 à 2. En augmentant s, nous augmentons la préférence pour utiliser certaines mêmes clés, ce qui augmente le taux de conflit. La Figure 5.12 trace le Goodput pour les solutions HLF, SyncMap, LockFree et MutexLock en utilisant différentes valeurs du paramètre s de la distribution Zipf. Nous pouvons voir que lorsque s augmente, le taux

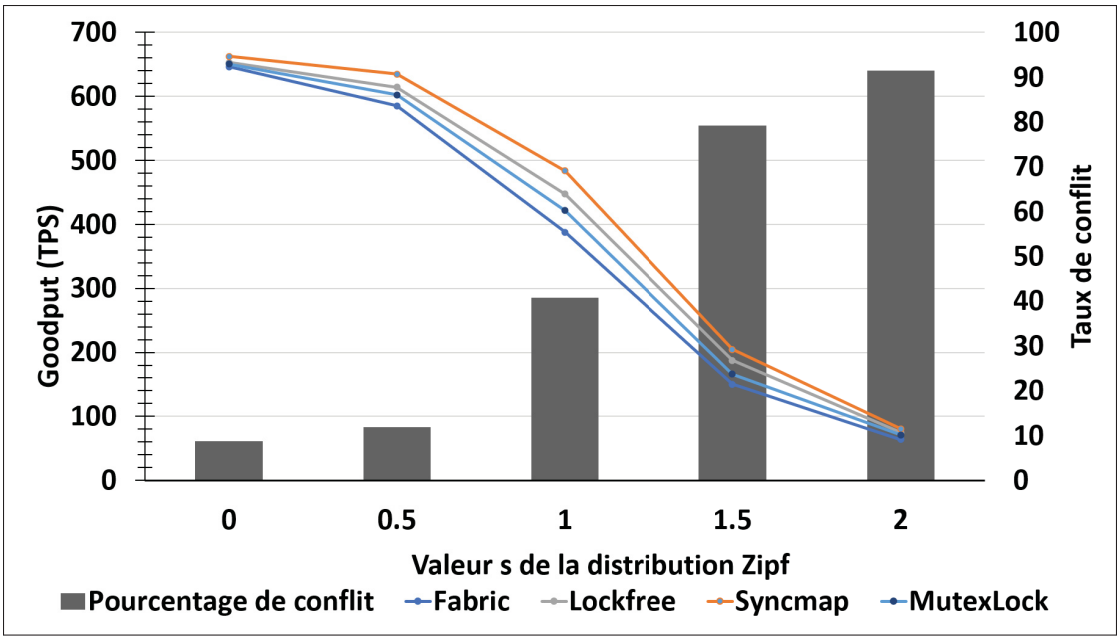


Figure 5.12 Impact de l'implémentation du chaincode

de conflit augmente, entraînant une diminution du Goodput. Comme prévu, lorsque le paramètre s augmente, nos solutions donnent des meilleures performances en comparant avec la référence. Par exemple, SyncMap améliore le Goodput de 24% pour *s* égal à 1 et de 35% pour *s* égal à 1.5.

5.5 Résumé et discussion des résultats obtenus

Nos trois solutions proposées sont plus performantes que la référence de HLF. SyncMap est la meilleure structure de données qu'on peut utiliser pour implémenter le cache. Le taux d'amélioration par rapport à HLF est principalement influencé par les politiques d'approbation utilisées, la topologie du réseau et l'implémentation du chaincode (taux de conflit). Dans des scénarios réalistes, où 40% des transactions sont rejetées à cause des conflits MVCC, notre solution SyncMap améliore le débit de 23% et réduit la latence de 80%.

La solution proposée est plus performante en utilisant la politique d'approbation AND avec un nombre maximum d'organisations et le minimum nombre de pairs par organisation possible. Cependant, lorsque la politique d'approbation utilisée est OR la solution est plus performante en

utilisant le nombre minimum d'organisations et de pairs par organisation possible. La taille du bloc doit être ajustée soigneusement dans le but de minimiser le taux de conflit de transactions interbloc et intrabloc pour maximiser le Goodput. Lors de l'implémentation du chaincode, il faut éviter au maximum les conflits entre les transactions afin d'assurer un taux de conflit inférieur à 40%.

5.6 Conclusion

Dans ce chapitre, nous avons comparé les trois solutions que nous avons implémentés à la référence de HLF originale. Nous avons aussi fourni une analyse de sensibilité afin d'étudier l'influence de certains paramètres sur les performances de nos solutions proposées. Enfin nous avons discuté les résultats obtenus et nous avons donné des recommandations lors de l'utilisation de nos solutions.

CHAPITRE 6

ÉTUDE DE CAS : ANALYSE DE PERFORMANCES DE HLF POUR UNE MONNAIE DIGITALE DE BANQUE CENTRALE (MDBC)

L'émergence des cryptomonnaies tel que Bitcoin et des systèmes de chaînes de blocs privés et publics à suscité un intérêt considérable des institutions financières vu les avantages importants offerts par cette technologie. Les banques ont commencé à mettre sur la table les projets de développement d'une monnaie digitale de banque centrale (MDBC) qui représente une alternative de la monnaie fiduciaire, mais sous forme numérique. Cette monnaie doit être émise seulement par la banque centrale et elle doit être régit par la réglementation gouvernementale, l'autorité monétaire ou la loi. Les MDBC pourraient être conçues avec les avantages de l'argent liquide sécurisé, accessible et privé, mais dans un format numérique que les gens pourraient utiliser pour effectuer des transactions électroniques, d'une personne à une autre, avec des commerçants dans des magasins et en ligne. La technologie des registres distribués telle que la chaîne de blocs est un outil qui permet d'implémenter une MDBC dans le but d'effectuer des transactions financières de manière transparente, plus rapide, sécurisée, et à moindre coût en comparant avec l'utilisation des technologies classiques qui serait plus efficace au service des finances et de l'économie en général.

6.1 Contexte du projet

Dans le but de mieux comprendre le comportement des réseaux HLF envers le changement des paramètres du réseau ainsi que la charge du travail, ce projet de conception, analyse et évaluation d'une monnaie digitale de banque centrale est élaboré au cours d'un stage de trois mois au sein de la Banque du Canada.

6.1.1 Problématique

Vu l'émergence des applications utilisant les systèmes de chaîne de blocs dans différents secteurs, les banques à travers le monde ont décidé de tirer profit de cette technologie et

ils ont lancé des programmes d'expérimentations de monnaie digitale de banque centrale. L'utilisation des systèmes de chaîne de blocs posent toujours des discussions autour de deux aspects très importants qui sont le débit effectif que le système peut assurer plus précisément pour des applications à temps réel tel que la MDBC, ainsi que la confidentialité des données des utilisateurs de l'application. Dans ce contexte, la banque du Canada voulait développer une preuve de technologie pour répondre aux questions sur la faisabilité d'une telle application sur les plateformes des registres distribués.

6.1.2 Objectifs

L'objectif principal de ce travail est la conception et l'implémentation d'une preuve de technologie (POT) d'une monnaie digitale de banque centrale en utilisant HLF dans le but de répondre aux questions suivantes :

1. Quelles sont les architectures et les caractéristiques possibles des différentes implémentations qui peuvent être faites ?
2. Quelles sont les limites de performances que le système peut offrir en termes de débit effectif et de latence pour les différences architectures proposées ?
3. Si les transactions sont conçues pour être privées à chaque organisation, quelles est l'impact sur les performances et la cohérence du réseau HLF ?

6.1.3 Méthodologie

Dans ce projet, nous avons limité notre étude au système HLF vu qu'il est un système de chaîne de blocs privé qui offre niveau élevé de gouvernance, de contrôle et de maintenance de réseau pour la banque. En plus, c'est un système qui peut garantir un débit et une rapidité d'exécution des transactions plus élevés que les autres systèmes de chaînes de blocs privées.

Afin d'accomplir ce travail, nous avons procédé selon les étapes suivantes :

1. Conception de l'architecture du réseau avec discussion sur le degré de confidentialité de chaque solution proposée

2. Implémentation du réseau HLF
3. Implémentation du chaincode (Contrat intelligent)
4. Évaluation des différentes solutions proposées

6.2 Revue de littérature

Durant les dernières années, les cryptomonnaies ont eu un développement important, mais cette monnaie numérique ne pourrait pas remplacer la monnaie fiduciaire vue sont aspect décentralisé et sans autorité de contrôle. Dans cette section, nous présentons les principaux travaux en relation avec l'implémentation des monnaies digitales de banque centrale.

La vague des MDBCs a commencé en 2011 (Sun *et al.*, 2018). Plusieurs banques centrales ont commencé à faire de la recherche et des preuves de concept, citons par exemple la banque d'Angleterre qui a commencé à explorer le MDBC et elle a proposé un modèle de monnaie digital appelé RSCoin (Danezis & Meiklejohn, 2015). Aussi la banque centrale hollandaise a développé un premier prototype de MDBC appelé DNBCoin basé sur bitcoin (Naudts, Aerts, franken & Pieterse, 2021). Plusieurs pays autres pays on aussi investit dans de tels projets tels que l'Inde (Wills & Sanders, 2017) et la Chine (China_Bank, 2021). Des analyses statistiques ont été élaborées par (Alonso, Jorge-Vazquez & Forradellas, 2021) et (Cunha, Melo & Sebastião, 2021) dans le but d'analyser le progrès en matière de MDBC dans les différents pays ayant des projets pilotes. Ces études ont conclu que le Bahamas, la Chine et l'Uragway sont les leaders au niveau de ses projets. Il y a aussi des pays qui avancent considérablement vers la mise sur le marché de leurs MDBC telle que la Lituanie et l'Estonie en Europe, Malysie en Asie, Brazil en Amérique du Sud et l'Afrique du sur dans le continent africain. De plus ces études ont aussi met l'accent sur les avantages et les inconvénients de la mise en place d'une MDBC, son implication et les défis majeurs rencontrés lors de son adoption.

D'autres travaux de recherche ont été élaborés dans ce contexte, en effet, (Han, Yuan & Wang, 2019b) proposent et décrivent une plateforme adéquate pour l'implémentation d'une MDBC assurant la sécurité requise pour une telle application. Aussi (Sun *et al.*, 2018) proposent un

modèle de MDBC appelé monnaie digitale à blockchain multiple basé sur les techniques de chaîne de blocs privée afin d'offrir de meilleures performances ou chaque banque commerciale détient sa chaîne de blocs appelée LAB (Local Area Blockchain), ainsi toutes les banques appartiennent à une blockchain contrôlée par la banque centrale du pays appelée Superchain. Un système hybride de blockchain pour l'implémentation d'une MDBC a été proposé par (Zhang *et al.*, 2021) basé sur le modèle balance pour les paiements massifs à petit montant et le modèle UTXO pour les montants ou les actifs ayant une valeur élevée.

6.3 Notions de bases

Dans cette section, nous présentons les deux modèles de données qu'on a comparés ainsi nous définissons les concepts de base qu'on a utilisés dans la conception des différentes architectures telle que les ZKAT, les mixeurs d'identité et les collections privées.

6.3.1 Modèle de données

Afin de stocker l'état de la chaîne de blocs les deux modèles de données UTXO et balance sont proposés. Ces deux modèles diffèrent essentiellement dans la manière avec laquelle ils sauvegardent l'état du système.

6.3.1.1 Modèle UTXO

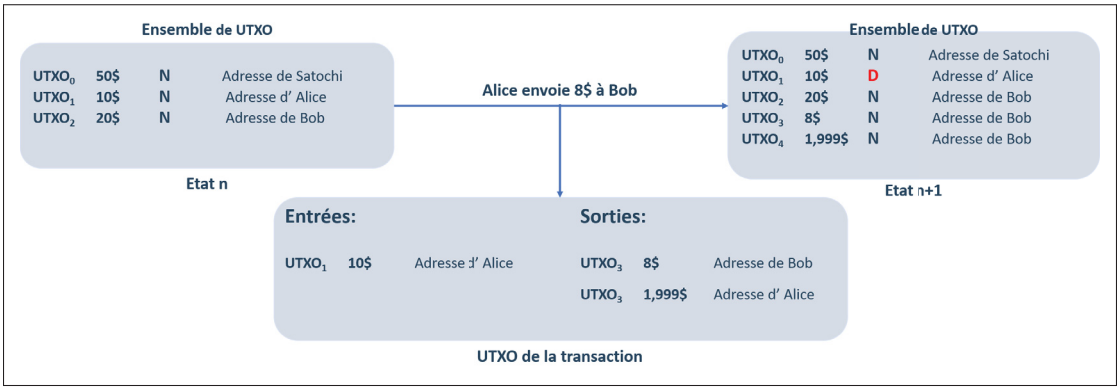


Figure 6.1 Modèle de transaction par UTXO

UTXO est l'abréviation du terme anglais *Unspent Transaction Output* (Pérez-Solà, Delgado-Segura, Navarro-Arribas & Herrera-Joancomartí, 2019). Ce modèle est basé sur l'historique des transactions (Ding *et al.*, 2019) où toutes les opérations sont stockées sous forme d'un graphe acyclique (DAG) entre des adresses de participants. Chaque transaction consomme des sorties non dépensées des transactions précédentes pour créer de nouvelles sorties qui peuvent être utilisées par des transactions au futur permettant d'assurer un niveau de confidentialité et de sécurité élevé par rapport au modèle balance (Kaneko, Osada, Azuchi, Okada & Yamasaki, 2019). La Figure 6.1 représente un exemple de transaction effectuée avec le modèle UTXO. Supposant qu'on a deux utilisateurs Alice et Bob ayant deux UTXO non dépensés (marqué par N sur la figure) de 10$ et 20$. Si Alice envoie 8$ à Bob, elle va utilisé $UTXO_1$ ayant une valeur de 10$ comme entrée de la transaction. Une fois la transaction est validée deux sorties vont être créer, la première avec l'adresse de Bob ayant une valeur de 8$ et la deuxième ayant la valeur 1.999$ représentant le reste de l'UTXO qui revient à Alice. La différence de 0.001 représente les frais de transaction qui vont être envoyés au mineur. À signaler que $UTXO_1$ devient une sortie de transaction dépensée (marqué par D sur la figure) qui ne pas être réutilisé.

6.3.1.2 Modèle Balance

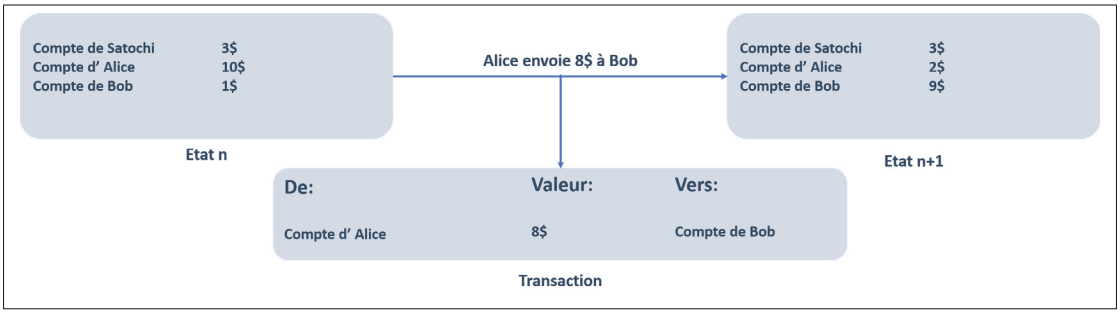


Figure 6.2 Modèle de transaction avec balance

Le modèle Balance permet de garder une trace des balances des comptes dans le système de chaînes de blocs. La Figure 6.2 représente un exemple de transaction en utilisant un modèle de balance. Si Alice envoie à Bob 8$, une opération de soustraction de ce montant va être exécutée si Alice a ce solde pour l'ajouter au compte de Bob.

6.3.1.3 Comparaison entre les deux modèles de données

On peut comparer les deux modèles selon deux critères principaux qui sont la taille de la blockchain et la confidentialité. Quant à la taille, le modèle de balance est plus efficace vu qu'il enregistre un seul solde de compte permettant d'économiser de la mémoire par rapport au modèle UTXO qui constitue le solde total d'un utilisateur à partir des UTXOs. Pour la confidentialité les deux modèles ont des avantages, le modèle UTXO rend le suivi de la propriété des actifs plus difficile par rapport au modèle de balance. Alors que le modèle de balance assure une meilleure fongibilité résultant de la transparence de tous les mouvements des actifs. Après cette analyse on a choisi d'utiliser le modèle de balance pour l'implémentation de la MDBC.

6.3.2 Zero-knowledge proof (ZKP)

Le concept ZKP permet à une partie qui possède une information (le prouveur) de prouver à une autre partie (le vérificateur) que l'information qui détient est correcte sans révéler l'information réelle. C'est une méthode de montrer que vous savez quelque chose qui satisfait une condition sans montrer ce que vous savez.

6.3.3 Zero-Knowledge Asset Transfer (ZKAT)

Ce concept a été présenté en 2018 par la communauté de HLF, ZKAT se base sur la technique de ZKP afin d'assurer la gestion des actifs sur le réseau de chaîne de blocs en préservant la confidentialité des données avec la possibilité d'audit sur le système. Ce concept permet au participant de créer et transférer des actifs sur le réseau HLF sans révéler des données en public concernant l'actif ou les détails de la transaction tout en respectant les réglementations du système.

6.3.4 Mixeur d'identité

Le mixeur d'identité est implémenté dans HLF à partir de la version 1.2. Il se base aussi sur le concept ZKP pour offrir la possibilité d'authentification anonyme des clients en exécutant les

transactions. En d'autres termes, le mixeur d'identité permet de cacher l'identité des parties prenantes de la transaction.

6.3.5 Les collections de données privées

Comme illustré dans la Figure 6.3, les collections de données privées sont composées de deux éléments principaux :

- **Les données privées** : les données privées sont échangées et stockées en claires dans des registres privés des organisations autorisées à accéder aux données
- **L'état haché du canal** : pour les organisations non autorisées à accéder aux données privées vont voir seulement le hash des données. Ce hash sert à endosser, ordonnancer et valider les transactions.

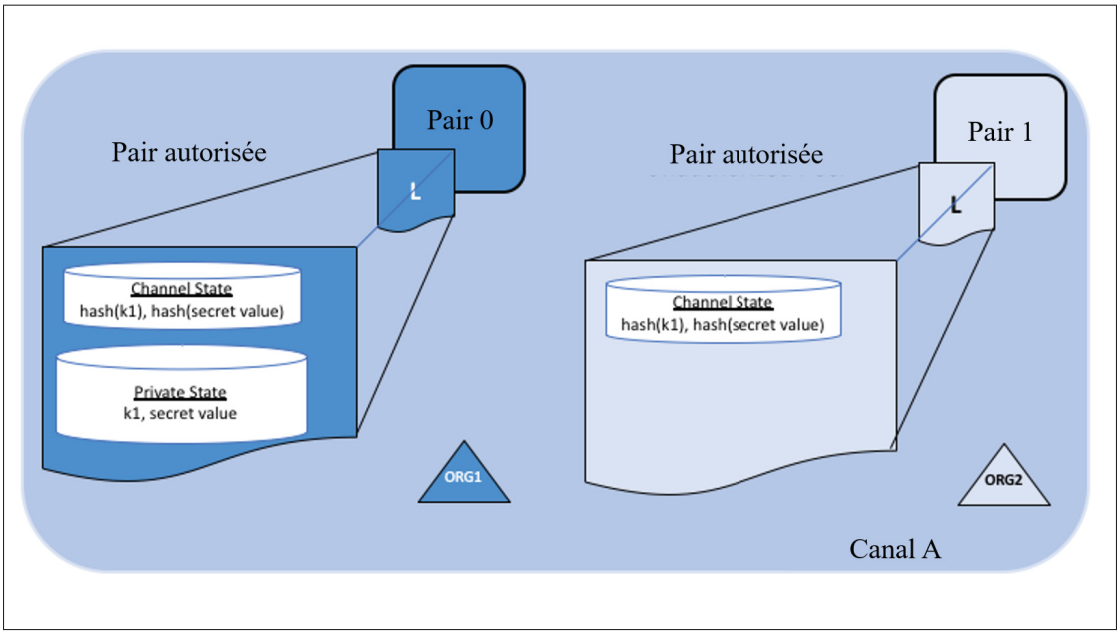


Figure 6.3 Collections de données privées

Les collections privées permettent d'assurer la confidentialité des données tout en préservant la cohérence et la sécurité de la chaîne de bloc. Tel qu'illustrer sur la Figure 6.5, les données sont échangées sous forme de hash ainsi que leur stockage haché sur les bases de données des

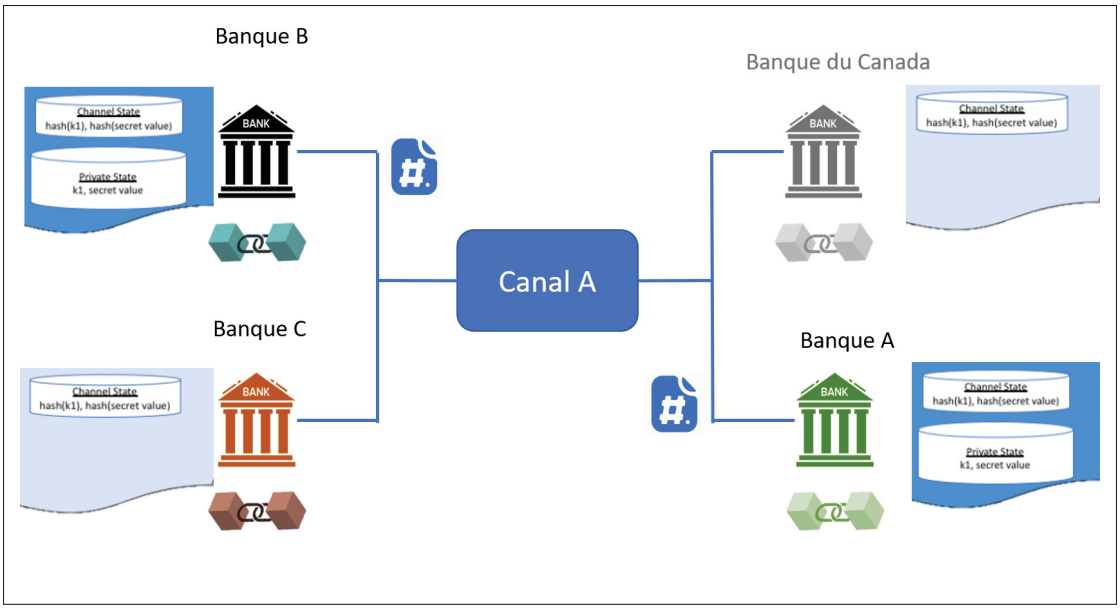


Figure 6.4 Architecture avec les collections privées

pairs non autorisés comme la banque C alors que sur le pair autorisé à accéder aux données le stockage se fait en claire comme les banques A et B.

6.3.5.1 Flux de transaction en utilisant des données privées

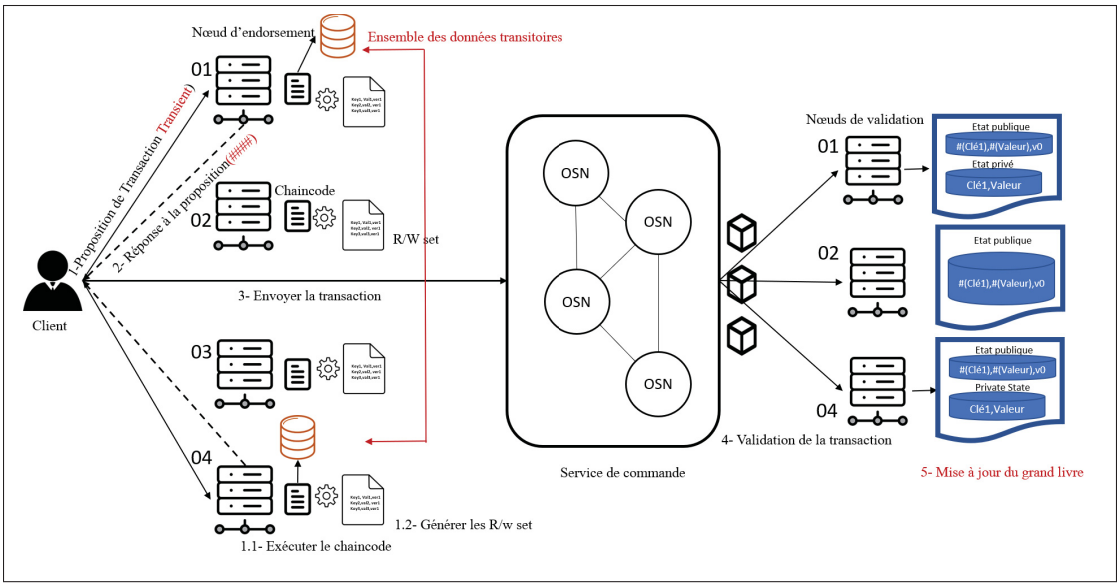


Figure 6.5 Flux de transaction avec des données privées

La Figure 6.5 représente le flux de transaction en utilisant les collections de données ainsi que les différences majeures avec le flux d'une transaction classique. Lorsque le client soumet une proposition de transaction, toutes les données sont stockées dans une entité transitoire qui sera supprimée lorsque la proposition de transaction est consommée par le pair endosseur. Ensuite, le chaincode est exécuté au niveau des pairs endosseurs et les données privées générées lors de la simulation sont stockés dans l'ensemble de données transitoires. Le pair envoie les données privées non chiffrés aux noeuds autorisés sur la base des politiques des collections à travers le protocole gossip. Le pair endosseur revoit une réponse à la proposition comprenant des données publiques, ainsi qu'un hachage de toutes les clés et valeurs de données privées. Par la suite, le client soumet la transaction aux services de commande qui l'ordonne dans un bloc et diffuse l'ensemble des transactions aux pairs de validation qui à leur tour peuvent valider les transactions en effectuant la vérification de la syntaxe, la validation VSCC qui est la validation des politiques d'approbation et la validation MVCC puisque la version correspondant à chaque clé est en claire. Après la validation, les pairs utilisent les politiques des collections pour déterminer s'ils sont autorisés à accéder aux données privées. S'ils ont le droit à le faire, ils vérifieront leur ensemble de données transitoires local et valideront les données privées avec le hachage sur le bloc et appliqueront l'ensemble lecture/écriture. Un pair non autorisé stockera simplement les données hachées sur leur état public.

6.3.5.2 Configuration d'une collection privée

Une définition de collection contient une ou plusieurs collections, chacune ayant une définition des organisations membre de la collection, ainsi que des propriétés utilisées pour contrôler la diffusion des données privées au moment de l'endossement et éventuellement, si les données seront purgées. Les définitions de collection sont composées des propriétés suivantes :

- **Name** : nom de la collection
- **Policy** : définit les pairs de l'organisation autorisés à conserver les données de collecte
- **RequiredPeerCount** : nombre de pairs requis pour diffuser les données privées

- **MaxPeerCount** : nombre de paires sur lesquelles les données privées sont dupliquées. Si un pair d'approbation tombe en panne, ces autres pairs sont disponibles au moment de la validation s'il y a des demandes pour extraire les données privées.
- **BlockToLive** : après ce nombre de blocs, la collection de données privées sera purgée.
- **MemberOnlyRead/Write** : une valeur *True* indique que seuls les clients appartenant à l'une des organisations membres de la collection sont autorisés à accéder en lecture/écriture aux données privées
- **endossementPolicy** : définit la politique d'approbationssement qui doit être respectée afin d'écrire à la collection de données privée.

```
[
 {
    "name": "collectionMarbles",
    "policy": "OR('Org1MSP.member', 'Org2MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive":1000000,
    "memberOnlyRead": true,
    "memberOnlyWrite": true
 },
 {
    "name": "collectionMarblePrivateDetails",
    "policy": "OR('Org1MSP.member')",
    "requiredPeerCount": 0,
    "maxPeerCount": 3,
    "blockToLive":3,
    "memberOnlyRead": true,
    "memberOnlyWrite":true,
    "endossementPolicy": {
```

```
        "signaturePolicy": "OR('Org1MSP.member')"
      }
  }
]
```

6.4 Conception

Dans le but de faire la conception de notre système, nous avons discuté le modèle de données qu'on va utiliser (UTXO/Balance) ainsi que les différentes architectures et techniques possibles qui permettent d'assurer un niveau élevé de confidentialité. Dans ce qui suit, nous allons présenter les différentes conceptions proposées.

6.4.1 Architecture de référence

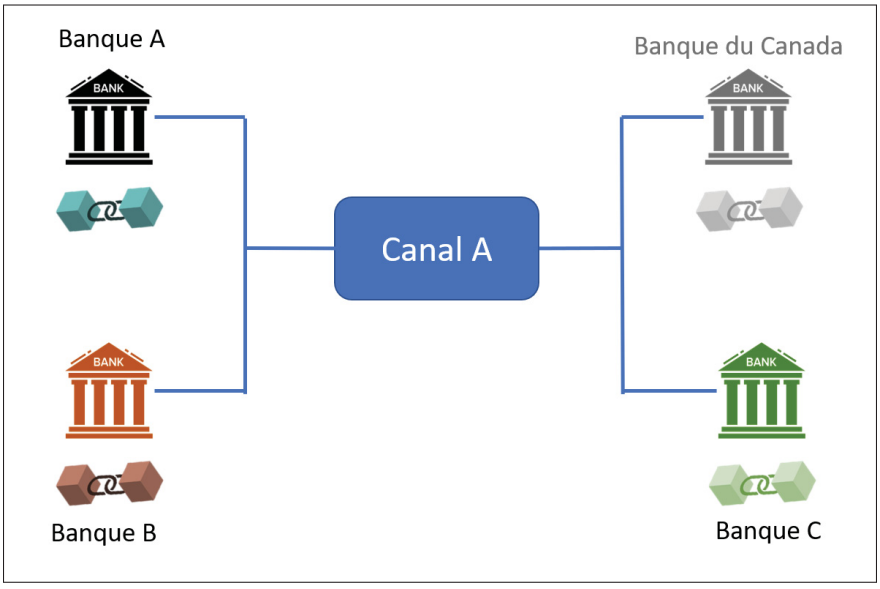


Figure 6.6 Architecture de référence

Comme architecture de référence, nous avons choisi un réseau HLF de base, tel que représenté sur la Figure 6.6, le réseau est constitué des banques commerciales et de la banque du Canada qui joue le rôle de l'entité de contrôle. Les participants communiquent sur un canal public ou les

données sont accessibles à tous les intervenants. Cette architecture est résistante au problème de doubles dépenses mais elle ne respecte pas la confidentialité des données bancaires.

6.4.2 Architecture avec des canaux privés bilatéraux

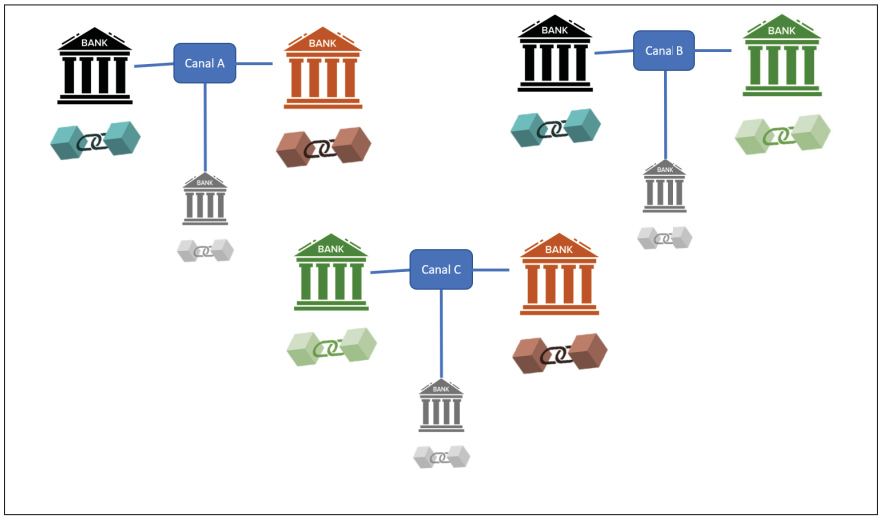


Figure 6.7 Architecture avec des canaux privés bilatéraux

La Figure 6.7 illustre l'architecture du réseau avec des canaux privés bilatéraux ou chaque deux banques commerciales communiquent sur un canal privé. Ce canal est accessible seulement pour les deux banques concernées ainsi que la banque centrale du Canada qui contrôle tous les canaux du réseau. Cette architecture assure un niveau élevé de confidentialité, en contrepartie elle n'est pas scalable puisqu'avec l'augmentation du nombre des organisations va augmenter exponentiellement le nombre de canaux.

6.4.3 Architecture avec la banque du Canada comme entité centrale

Cette architecture consiste à déployer un réseau HLF ou la banque du Canada va être l'entité centrale du réseau à travers toutes les transactions vont passé tel qu'illustré dans la Figure 6.8. Si une la *Banque A* veut envoyer des données à la *banque B*, elle doit obligatoirement passé par la BC. Cette architecture assure la confidentialité vu que la BC va gérer le partage de données ainsi elle permet d'avoir une relation linéaire entre le nombre de canaux et d'organisations mais elle

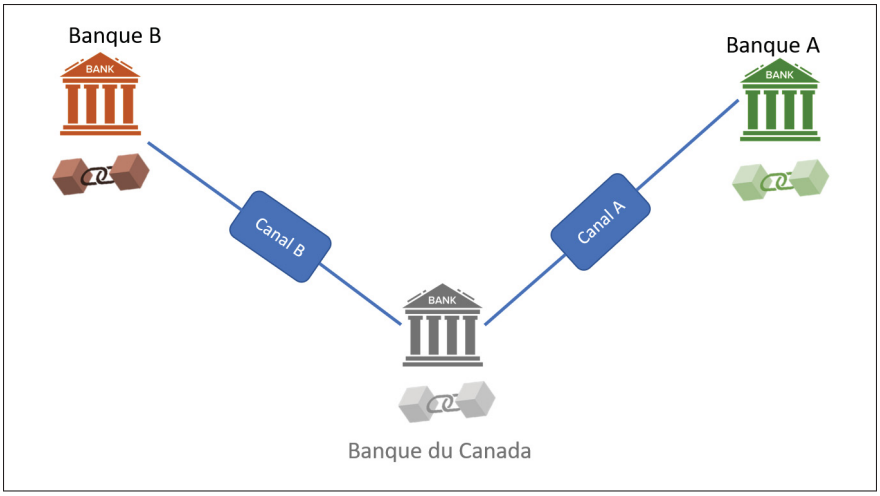


Figure 6.8 Architecture avec la BC comme entité centrale

nécessite un transfert d'actifs entre les canaux qui va augmenter la latence des transactions et qui n'est pas encore implémenté dans HLF jusqu'à la version 2.2 qu'on a utilisé pour ce projet.

6.4.4 Architecture de référence avec ZKAT et Mixeur d'identité

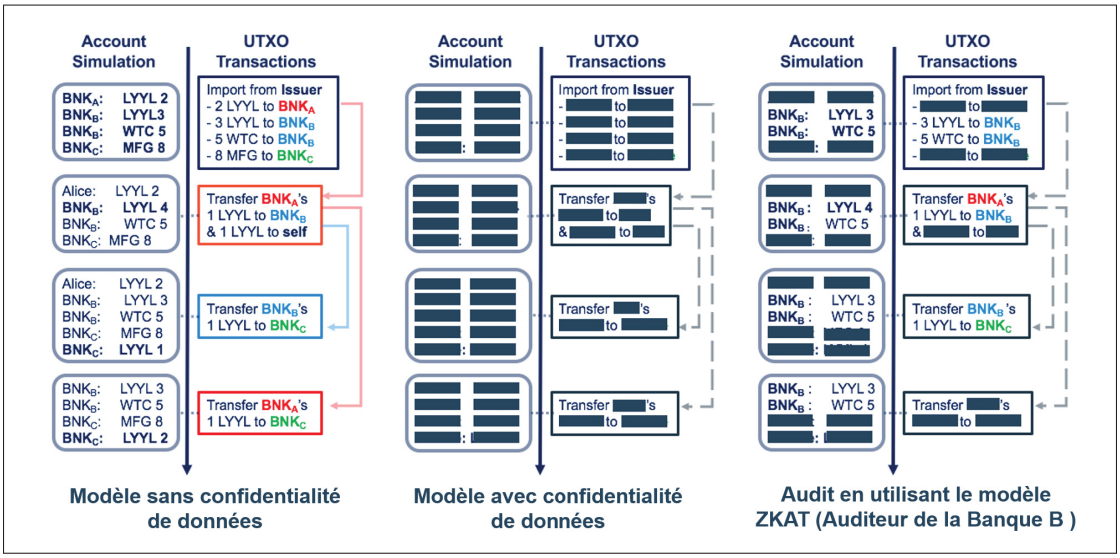


Figure 6.9 Architecture de référence avec ZKAT et Mixeur d'identité

La quatrième architecture qu'on a proposée est basée sur les concepts ZKAT et mixeur d'identité. Au lieu de faire les transferts d'actifs en public, on utilise une fonction ZKP pour crypter les données et l'envoyer à la banque concernée par la transaction. Dans ce cas, les données peuvent être vérifiées en utilisant la preuve ZKP sans révéler les données que ce soit à la phase d'endossement ou de validation. Ceci peut être combiné avec un mixeur d'identité dans le but de cacher les identités des clients concernés par la transaction. La Figure 6.9 représente l'impact des deux techniques sur les données au niveau du registre distribué. Le modèle à gauche est un modèle sans confidentialité ou toutes les données sont publiques. Celui au milieu illustre l'utilisation du ZKAT qui cache les informations liées à l'actif échangé et du mixeur d'identité qui permet de masquer les identités des participants. Le modèle à droite met en évidence un cas d'audit que les ZKAT peut offrir pour la banque B, dans ce cas la banque peut voir toutes les transactions dans lesquelles elle été impliquée.

Malheureusement, on n'a pas pu utiliser cette approche parce que le ZKAT est décrit dans l'aperçu technique de HLF, mais il n'est pas encore implémenté jusqu'à la version 2.2 de HLF.

6.4.5 Architecture avec une collection privée par banque

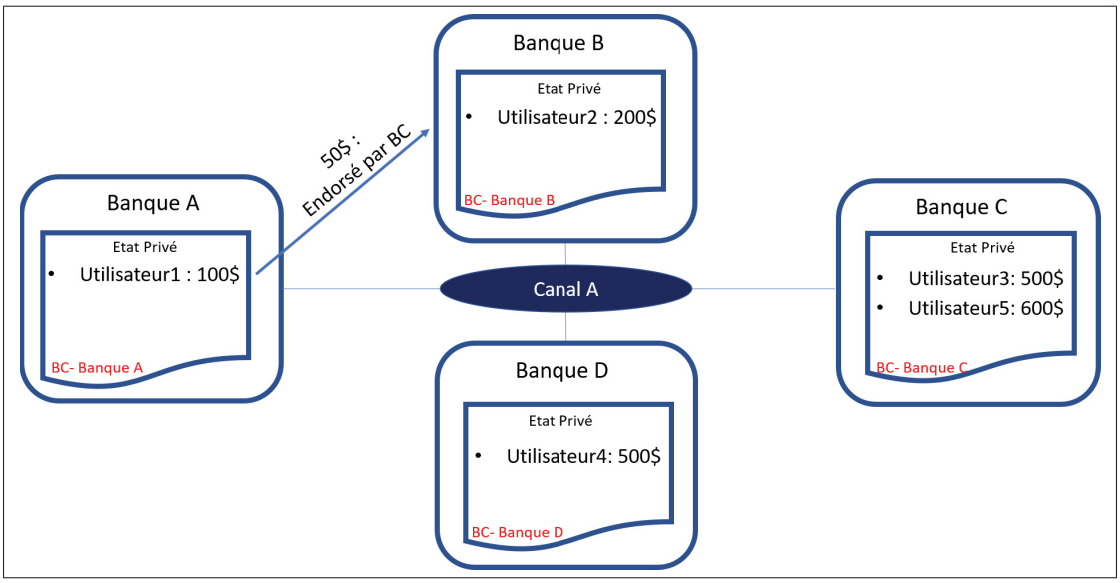


Figure 6.10 Architecture avec une collection privée par banque

Pour cette approche, nous mettrons en œuvre une collection de données privée pour chaque banque où le la collection est accessible uniquement par cette banque et la banque du Canada. Avec cette solution si l' *utilisateur 1* de la banque A veut envoyer de l'argent à l'*utilisateur 2* de la banque B, la transaction doit être approuvée par la banque du Canada car c'est la seule entité qui peut accéder aux deux collections privées, cependant, si l'*utilisateur 3* de la banque C envoie de l'argent à l'*utilisateur 5* de la même banque, la transaction peut être approuvée par la banque C ou la Banque du Canada. Cette solution assure un niveau élevé de confidentialité des données mais elle risque de surcharger la Banque du Canada car elle avalisera beaucoup de transactions sur le réseau.

6.4.6 Architecture avec des collections privées partagées et collections privées par banque

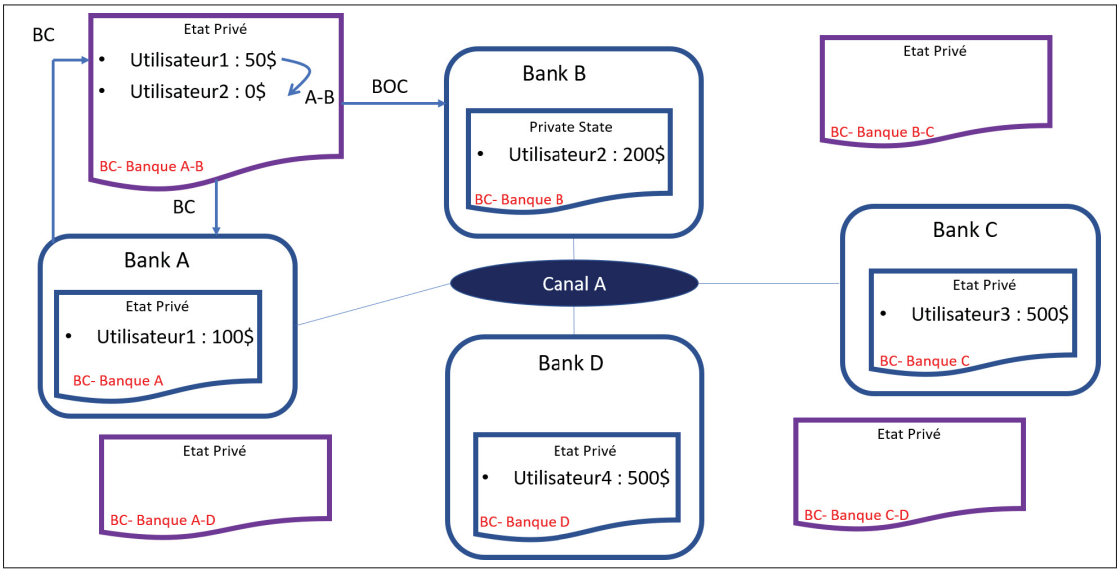


Figure 6.11 Architecture avec des collections privées partagées et collections privées par banque

Afin d'alléger la charge sur la Banque du Canada, nous pouvons utiliser des collections privées et partagées. Dans ce modèle, nous aurons une collection privée pour chaque banque et une collection partagée entre chaque deux banques avec la Banque du Canada, donc si un *utilisateur 1* de la banque A envoie 50 $ à un *utilisateur 2* de la banque B, la transaction se fait en deux

étapes : d'abord, la transaction est effectuée pour transférer 50$ de la collection de la banque A vers la collection partagée avec la banque B. Cette transaction doit être approuvée par la Banque du Canada, puis le transfert est effectué dans la collection partagée, et la transaction est approuvée par la banque A et la banque B. Chaque fois que la banque souhaite agréger le solde de l'utilisateur, elle peut le faire en transférant le solde de l'utilisateur de la collection partagé vers sa collection privée et cette transaction doit également être approuvée par la Banque du Canada. Ainsi, cette solution allégera la charge sur la Banque du Canada s'il y a beaucoup de transactions entre les utilisateurs de différentes banques.

6.4.7 Architecture passant toutes les transaction à travers la banque du Canada

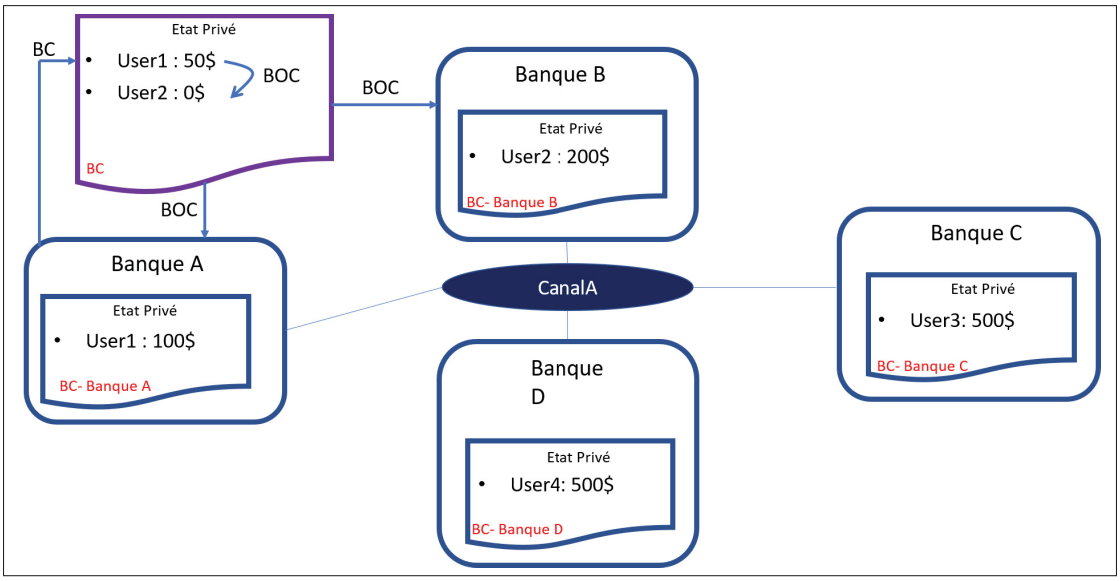


Figure 6.12 Architecture passant toutes les transaction à travers la banque du Canada

Pour cette approche, nous mettons en place une collection de données privée pour chaque banque ainsi qu'une collection pour la Banque du Canada. Avec cette implémentation, si un *utilisateur 1* de la banque A veut envoyer de l'argent à un *utilisateur 2* de la banque B, la transaction est effectuée en transférant 50$ de la collection de la banque A à la collection de la Banque du Canada. Cette transaction doit être approuvée par la Banque du Canada car elle est la seule entité qui peut accéder aux deux collections privées. Ensuite, le transfert est effectué dans la collection

privée de la Banque du Canada et la transaction est également approuvée par la Banque du Canada. Après cela, les soldes peuvent être agrégés dans la collection de la banque commerciale. Cette solution surchargera la Banque du Canada par rapport à l'approche précédente puisqu'elle a plus de transactions à approuver.

6.4.8 Architectures analysées

Vue la contrainte de temps pendant le stage, on a décidé d'implémenter l'architecture de référence ou toutes les données sont sur un seul canal partagé et l'architecture avec une collection privée par banque dans le but de comparer les performances de l'architecture de référence a celle avec les collections de données privées pour savoir le coût en termes de performances d'augmentation du niveau de confidentialité.

6.5 Évaluation des performances

Dans ce chapitre, nous commençons par détailler l'environnement d'implémentation de notre réseau HLF. Ensuite, nous allons fournir une analyse comparative entre l'architecture de référence et l'architecture avec une seule collection par banque commerciale.

6.5.1 Environnement d'évaluation

Décrivons la configuration avant de commencer l'interprétation des résultats expérimentaux. Tel qu'illustré dans la Figure 6.13, notre système d'évaluation se compose de neuf machines virtuelles hébergées sur le nuage Azure. Cinq machines servent comme des pairs appartenant à des organisations différentes et trois machines comme des noeuds du service de commande avec un algorithme de consensus Raft. La neuvième machine virtuelle joue le rôle des clients en exécutant Hyperledger Caliper qui est un outil d'évaluation des réseaux de chaîne de blocs pour plusieurs plates-formes telles qu'Ethereum et un ensemble de projets Hyperledger, y compris HLF. Chaque machine virtuelle possède 8 vCPU et 32 Go de RAM à part la machine hébergeant Caliper qui à 32vCPU pour nous permettre d'envoyer un nombre élevé de transaction dans le but

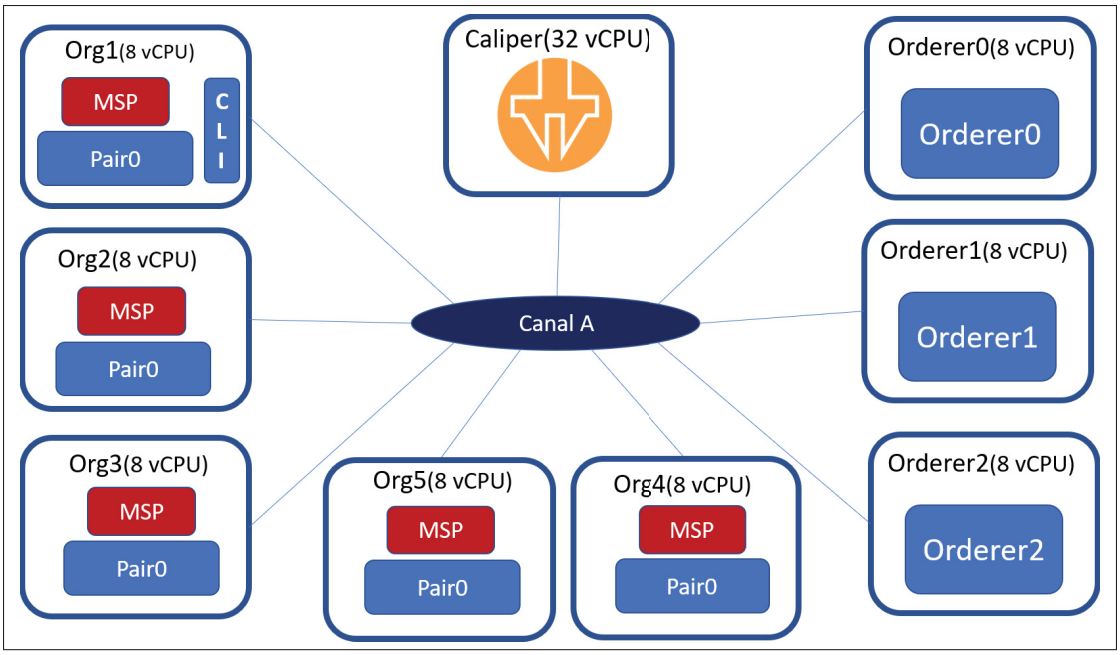


Figure 6.13 Environnement d'évaluation

d'améliorer le taux d'envoi des transactions. Toutes les machines virtuelles exécutent Ubuntu 16.04 en tant que système d'exploitation. Les pairs sont configurés pour utiliser CouchDB comme base de données. Pour le chaincode, nous avons développé chaincode de test qui implémente le standard ERC20.

6.5.2 Paramètres par défaut du réseau

Tableau 6.1 Paramètres par défaut du réseau d'évaluation

Paramètres	Values
Taille du bloc	300 TX
Temps du bloc	2s
Nombre d'organisation	5
Nombre de pairs par organisation	1
Consensus	Raft (3 OSN)
politique d'approbationssement	AND(Org1,Org2,Org3,Org4,Org5)

Le tableau 6.1 montre les paramètres utilisés par défaut pour notre expérience. Pour les paramètres du service des commande, on a utilisé un temps de bloc de 2s et une taille de bloc de 300 TX . Nous avons choisi le consensus Raft plutôt que le mode solo qui n'est pas recommandé qu'en cas de développement et de test (Wang & Chu, 2020) et le mode Kafka qui est obsolète dans les versions 2.x de Fabric. Nous avons également choisi 3 nœuds de services de commande pour satisfaire le compromis entre le coût du réseau déployé sur le nuage électronique et ses performances.

6.5.3 Évaluation des performances de l'architecture de référence

Nous utilisons l'architecture de référence afin de faire l'analyse du comportement du réseau avec le changement de quelques paramètres tels que la taille du bloc et le nombre de nœuds du service de commande.

6.5.3.1 Impact de la taille du bloc

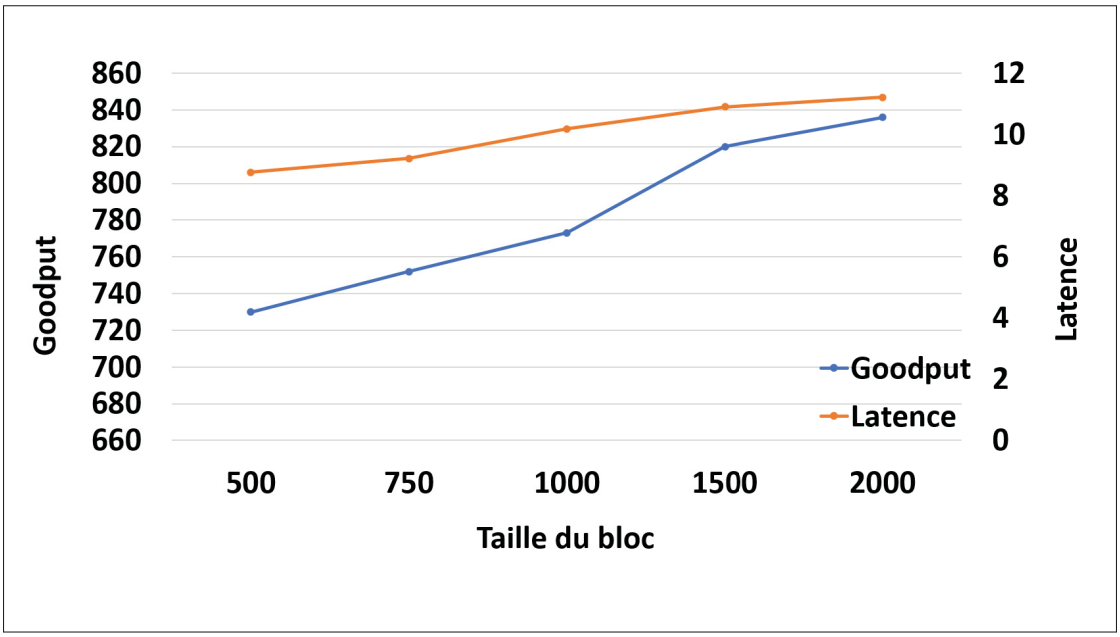


Figure 6.14 Impact de la taille du bloc

Dans cette section nous étudions l'impact de la taille du bloc sur les performances du réseau. La Figure 6.14 représente l'évolution du goodput en fonction de la taille du bloc. Nous constatons qu'avec une augmentation de la taille du bloc le débit augmente. Ceci est dû au fait que l' overhead diminue lorsque le bloc contient un nombre plus élevé de transactions. On remarque aussi l'augmentation de la latence avec l'augmentation de la taille du bloc qui est due au délai supplémentaire écoulé lors de la création du bloc au niveau du service des commandes.

6.5.3.2 Impact du nombre des noeuds de commande

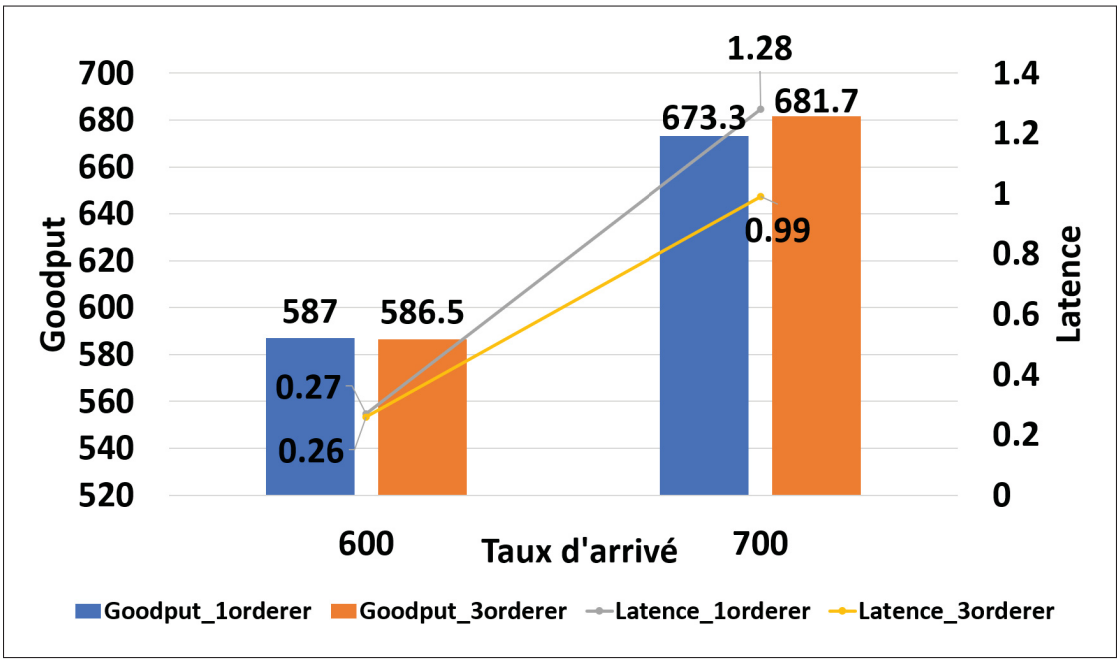


Figure 6.15 Impact du nombre des noeuds de commande

La Figure 6.15 trace le débit et la latence moyens pour divers temps de bloc tout en variant le taux d'arrivée des transactions et le nombre des noeuds de commande sur le réseau. Nous remarquons qu'avec une augmentation du nombre des noeuds de commande il n'y a pas d'impact sur le débit même en variant le taux d'arrivée des transactions. Avec une augmentation du nombre des noeuds de commandes, la latence diminue lorsque le taux d'arrivée augmente. Cette latence est expliquée par le fait que les services de commande sont devenus le goulot d'étranglement

du réseau à un taux d'arrivée des transactions élevé. Si le nombre de nœuds de service de commande augmente la charge sera équilibrée résultant la diminution de la latence.

6.5.3.3 Évaluation globale du système pour des applications à temps réel

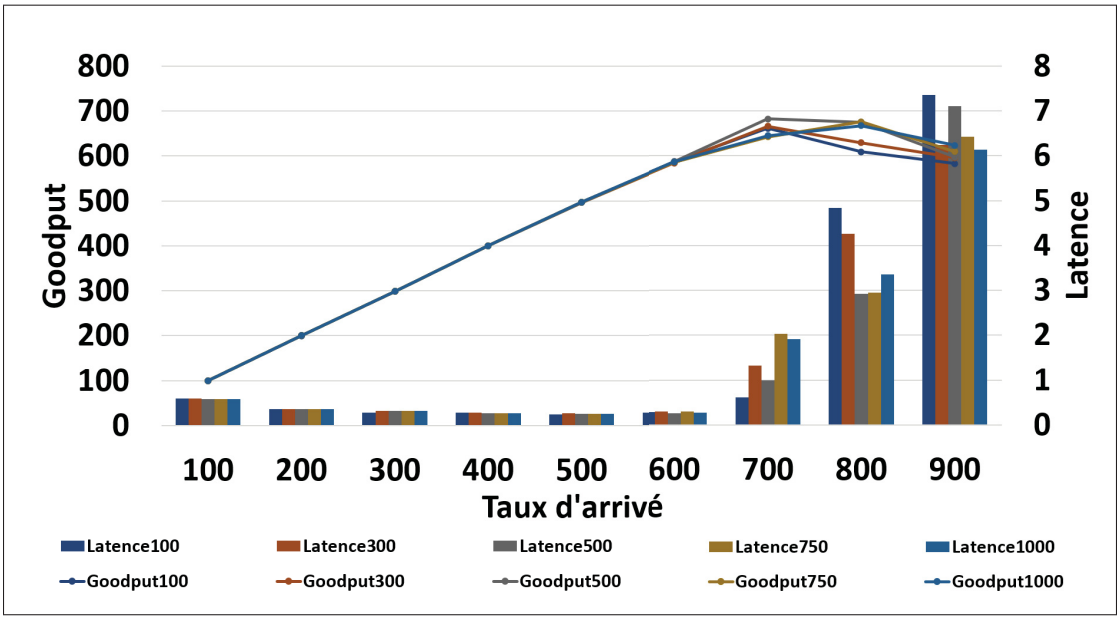


Figure 6.16 Évaluation globale du système pour des applications à temps réel

Pour notre cas d'utilisation du MDBC, nous avons besoin d'une latence inférieure à 1 seconde. La question qui se pose : quel est le débit que nous pouvons atteindre tout en assurant une latence inférieure à 1s ? Ainsi, nous définissons le temps de bloc comme la valeur optimale de 1s et observons l'impact de la variation de la taille du bloc. La figure 6.16 représente le débit effectif moyen et la latence pour différentes tailles de bloc sur différents taux d'arrivée des transactions. Avec une augmentation du taux d'arrivée des transactions, le débit a augmenté de manière linéaire comme prévu jusqu'à ce qu'il atteint un point de saturation à environ 650 tps. Lorsque le taux d'arrivée était proche ou supérieur à ce point, la latence augmente de manière significative (de l'ordre de 10 ms à l'ordre de quelques secondes).

En conclusion, si nous voulons un système qui offre une latence inférieure à une seconde, notre taux d'arrivée moyen ne doit pas dépasser environ 600 TPS.

6.5.4 Évaluation des performances des collections de données privées

Dans cette section, nous étudions l'évolution du débit effectif et de la latence du modèle d'une seule collection de données privées par banque décrit dans la Section 6.4.5.

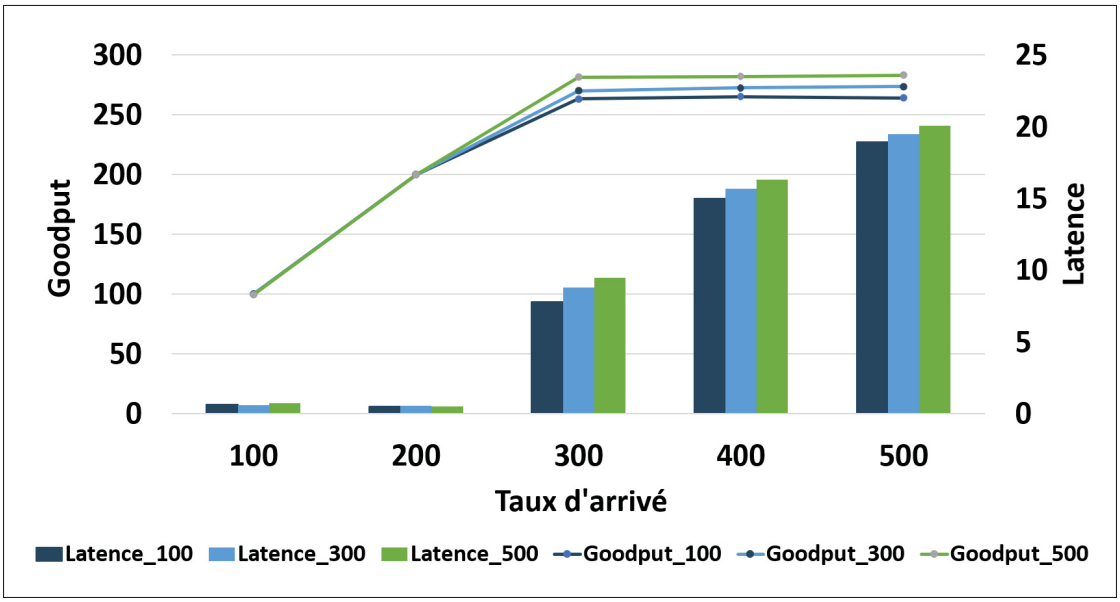


Figure 6.17 Évaluation globale du système avec des collections privées pour des applications à temps réel

La Figure 6.17 représente l'évolution du débit et de la latence pour différentes tailles de blocs et à différents taux d'arrivée de transaction.

Avec une augmentation du taux d'arrivée des transactions, le débit a augmenté de manière linéaire jusqu'à ce qu'il se stabilise à environ 270 tps (40% du débit atteint avec la référence). Lorsque le taux d'arrivée était proche ou supérieur à ce débit, la latence augmentait considérablement.

Cette diminution du débit et augmentation de la latence peut s'expliquer par le fait que la phase d'endossement ou de validation peuvent être un goulot d'étranglement, cela peut aussi être par le fait que la banque du Canada est surchargée vu qu'elle doit HLF la totalité des transactions.

6.6 Conclusion

Dans ce chapitre, nous avons discuté deux défis majeurs de l'adoption de la technologie de chaîne de blocs qui sont la confidentialité et les performances du réseau sur un cas d'utilisation de monnaie digitale de banque centrale. Nous avons proposé et discuté plusieurs modèles de données et architectures du réseau dans le but de résoudre le compromis entre la confidentialité et les performances du réseau. Ensuite, nous avons fourni une étude comparative entre un modèle de référence basique et un modèle utilisant les collections de données privées qui permettent d'assurer la confidentialité des données.

CHAPITRE 7

EXPÉRIENCE PERSONNELLE

Dans ce chapitre, je vais présenter mon parcours au cours de l'élaboration de ce mémoire en technologie de l'information à l'École de technologie supérieure. Au début je vais parcourir les étapes suivies qui ont permis de définir notre problématique de recherche ainsi que de fixer et atteindre les objectifs de ce mémoire. Ensuite, je vais énumérer les défis rencontrés dans le but de réaliser les objectifs de ce projet.

L'expérience de stage à l'École de technologie supérieure au cours de la session d'été 2019 m'a permis d'avoir une vue globale du processus de la maîtrise en recherche à l'ÉTS grâce à mes collègues du FUSÉE LAB. Ceci a facilité mon intégration à mon retour à Montréal pour commencer ma maîtrise en Hiver 2020. En effet, la première étape était l'identification de la problématique générale de recherche ensuite j'ai fait le choix de cours qui permettent d'atteindre mon objectif et de définir exactement le plan de la maîtrise. Le cours de la lecture dirigée a été essentiel pour faire la revue de littérature et parcourir les travaux de recherche scientifique optant à optimiser les performances de HLF, identifier les lacunes existantes au niveau des solutions proposées et formuler la solution qu'on va proposer et implémenter. À cette phase de j'ai repérer quelques défis a résoudre tel que le manque de la documentation sur un tel système de blockchain qui engendre de longues périodes de recherche sur un détail du fonctionnement réseau ou parfois faire des tests sur un réseau réel pour comprendre le comportement du réseau. À cet effet, j'ai réussi a définir en détail ma proposition de recherche qui consiste a faire l'optimisation des performances du réseau HLF par la détection précoce des conflits MVCC à la phase d'endossement au lieu de faire traitement d'une transaction conflictuelle tout au long du réseau alors qu'elle va finir par échoué à la phase de validation permettant d'améliorer le débit et de minimiser la latence des transactions sur le réseau.

Vu que j'ai une expérience de deux ans de développement avec HLF, j'ai directement entamé la formulation de la problématique à travers la modélisation du système étudié et la définition des formules mathématiques pertinentes. En utilisant ces formules, nous avons pu faire une

analyse théorique de la solution proposée afin d'avoir une estimation de quelques métriques de performance de la solution qui vont être validées par l'évaluation pratique du système proposé.

Au début de la session d'automne 2020, j'ai eu l'opportunité de faire un stage au sein de la banque du Canada pour faire la conception, l'implémentation et l'évaluation d'un système de monnaie digitale de banque centrale en utilisant HLF. Ce stage est la continuité de mon mémoire. En effet, il était l'occasion pour découvrir et d'utiliser Hyperledger Caliper qui est l'outil d'évaluation des performances des réseaux HLF. Cette opportunité m'a permis d'intégrer le milieu professionnel et d'acquérir des compétences sur le plan personnel et technique en faisant de la recherche sur un sujet de pointe. Pendant ce stage j'ai fait face à plusieurs défis à savoir la proposition des différentes architectures de réseau, le déploiement d'un réseau HLF distribué sur de différentes machines hébergées sur le nuage vu qu'auparavant j'ai utilisé docker pour déployer le réseau sur la même machine. En plus, l'apprentissage de l'utilisation de Hyperledger Caliper n'était pas assez facile vu que le projet est en phase de développent et la documentation n'était pas mature.

Après avoir terminé le stage, je reviens à me concentrer sur le projet de recherche en explorant le code source de HLF, cette phase m'a permis de parcourir les fichiers ainsi que les fonctions qui vont être altérés pour implémenter la solution proposée. Comme première étape d'implémentation, j'ai mis en place la structure de filtrage MVCC au niveau de la phase d'endossement ensuite j'ai développé les tests unitaires qui permettent de valider le fonctionnement du module EMVCC. Lorsque, j'ai implémenté la fonction qui permet de mettre à jour la structure EMVCC au niveau de la phase de validation, j'ai identifié le problème d'accès concurrent à la structure de données EMVCC. À cette étape j'ai exploré les solutions qui permettent de résoudre ce problème tel que l'implémentation d'un système de verrouillage (MutexLock) ou des systèmes sans verrouillage (LockFree et SyncMap). À ce moment j'ai décidé d'implémenter ces trois structures de données et de les comparés avec la référence de HLF. Enfin, j'ai passé à l'étape d'évaluation où j'ai comparé nos solutions à la référence de HLF et j'ai fait une étude de sensitivité pour évaluer l'impact de certains paramètres sur les performances des applications. Les résultats des expériences obtenues confirment les résultats théoriques ce qui valide la modélisation ainsi que l'implémentation.

Pour conclure, cette maîtrise était une occasion de développer mes compétences sur le plan personnel et professionnel. Cette expérience m'a permis d'approfondir mes connaissances techniques dans le domaine des chaînes de blocs. Par ailleurs, je compte commencer ma carrière professionnelle dans ce domaine qui me passionne que je trouve prometteur dans les prochaines trois à cinq années. De plus, ma maîtrise est aussi la porte pour découvrir le monde de la recherche scientifique ce qui me permet d'enrichir mon profile d'ingénieur. En effet, la recherche m'a éduqué par rapport à la manière avec laquelle on peut innover en analysant l'existant ainsi de développer un esprit critique permettant d'avoir un raisonnement rigoureux. Enfin, j'ai acquis aussi des compétences en termes de communication, de rédaction et de travail en équipe durant mes échanges avec mon superviseur et mes collègues dans différents secteurs de recherche.

CONCLUSION ET RECOMMANDATIONS

La chaîne de blocs est une technologie prometteuse vue les avantages qu'elle offre en termes d'immutabilité et de sécurité. Malgré le développement des systèmes de chaîne de blocs, le débit des transactions présente toujours un défi qui doit être amélioré pour avoir des valeurs comparables à celle des systèmes de base de données actuelles.

Dans ce mémoire, en premier lieu, nous présentons le volet principal qui est de concevoir une solution pour optimiser les performances de HLF par la détection précoce des transactions en conflit à la phase d'endossement au lieu de les traiter jusqu'à la phase de validation causant le ralentissement du débit des transactions et l'augmentation de la latence. Nous présentons trois solutions qui diffèrent principalement par leurs structures de données permettant le stockage des clés en écriture par les transactions en cours d'exécution. Nous avons élaboré une formulation du problème suivi par une analyse théorique du système. Ensuite, nous avons implémenté nos solutions en faisant des changements sur le code source de HLF. Nous évaluons les performances des solutions proposées sur un réseau HLF distribué sur le nuage informatique. Les résultats montrent que nos trois solutions ont des performances qui surpassent celle de la référence de HLF. La meilleure solutions proposées est SyncMap offrant le meilleur débit et la latence la plus faible. En deuxième lieu, nous conduisons une étude de cas en utilisant HLF pour implémenter et évaluer une application de monnaie digitale de banque centrale avec la version originale de HLF.

Notre meilleure solution peut être encore optimisée dans le but de minimiser le nombre de faux positive et des faux négatives. Comme travaux futurs, nous proposons l'amélioration de notre solution en trouvant une procédure de filtrage plus intelligente pour assurer une décision plus précise concernant la validité de la transaction résultant l'amélioration des performances du système.

ANNEXE I

EARLY DETECTION FOR MULTIVERSION CONCURRENCY CONTROL CONFLICTS IN HYPERLEDGER FABRIC

Helmi Trabelsi[1] , Kaiwen Zhang[1]

[1] Département de génie logiciel et des TI, École de technologie supérieure,
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

Article publié à la conférence « Crypto Valley Conference 2021. (CVC, Rotkreuz, Switzerland) »
(Accepté le 11 Octobre 2021).

Early Detection for Multiversion Concurrency Control Conflicts in Hyperledger Fabric

Helmi Trabelsi
Department of Software and IT Engineering
ÉTS Montréal
University of Québec
helmi.trabelsi.1@ens.etsmtl.ca

Kaiwen Zhang
Department of Software and IT Engineering
ÉTS Montréal
University of Québec
kaiwen.zhang@etsmtl.ca

***Abstract*— Hyperledger Fabric is a popular permissionless blockchain system that features a highly modular and extensible system for deploying permissioned blockchains which are expected to have a major effect on a wide range of sectors. Unlike traditional blockchain systems such as Bitcoin and Ethereum, Hyperledger Fabric uses the EOV model for transaction processing: the submitted transactions are executed by the endorsing peer, ordered and batched by the ordering services, and validated by the validating peers. Due to this EOV workflow, a well-documented issue that arises is the multi-version concurrency control conflict. This happens when two transactions try to writes and read the same key in the ledger at the same time. Existing solutions to address this problem includes eliminating blocks in favor of streaming transactions, repairing conflicts during the ordering phase, and automatically merging the conflicting transactions using CRDT techniques.**

In this paper, we propose a novel solution called *Early Detection for MVCC Conflicts*. Our solution detects the conflicting transactions at an early stage of the transaction execution instead of processing them until the validation phase to be aborted. The advantage of our solution is that it detects conflict as soon as possible to minimize the overhead of conflicting transaction on the network resulting in the reduction of the end-to-end transaction latency and the increase of the system's effective throughput. We have successfully implemented our solution in Hyperledger Fabric. We propose three different implementations which realize early detection. Our results show that our solutions all perform better than the baseline Fabric, with our best solution SyncMap which improves the goodput by up to 23% and reduces the latency by up to 80%.

I. INTRODUCTION

Hyperledger Fabric is a popular permissionless system that allows the development of blockchain applications in a variety of domains such as finance, healthcare, and supply chain management. Compared to Ethereum [3] and Bitcoin [13], Fabric offers superior performance, due to the consensus algorithm used and the small number of peers on the network compared to public blockchains. Furthermore, Fabric provides a flexible framework for managing responsibilities across parties using the membership service provider (MSP) which is an abstract component that maintains the identities and roles of all nodes that belong to the same organization.

Most existing blockchain platforms such as Ethereum and Hyperledger Fabric allow Turing-complete computations by executing a smart contract for a given transaction [21]. To guarantee consistency across the network, peers execute transactions to generate the next state of the blockchain after the content of the next block is known (which transactions to execute, and in what order): this is called the Order-Execute (OX) transaction flow [2]. The weaknesses of the OX pattern is that the sequential execution of the transactions within each block limits the throughput of the system. Furthermore, each peer requires knowledge of all smart contracts to compute the next state of the blockchain, which may present confidentiality and privacy issues.

Instead, Fabric employs the Execute-Order-Validate (EOV) pattern, where the transactions are executed in a sandbox called endorsing peers to generate the read-write sets including the versions of the used keys for the transaction's simulation, then the transactions are ordered by the ordering services into blocks, and at the end validated and committed to the ledger. EOV overcomes the limitations of OX by providing parallelism of transactions execution on different endorsing peers.

The drawback of the EOV pattern is that a read-write lock is used to synchronize the execution and validation phases. To solve this issue, Hyperledger Fabric implements a Multi-Version Concurrency Control mechanism to guarantee the consistency of the blockchain. When validating transactions by the validating peers, the versions of the generated read-write sets are compared to the keys' versions of the ledger to avoid that two transactions attempt to modify/read the same key/value pairs at the same time. A transaction will be rejected if its read set contains an old version of a key and the client has to resubmit it.

The aforementioned MVCC conflict problem is a well-documented limitation of Hyperledger Fabric as it decreases the effective throughput of the system (also known as goodput). This is because the blocks could contain aborted transactions which still count against the size limit, thereby wasting valuable block

space. These aborted transactions must also be retried by their respective clients as brand new transactions, which generates additional load on the system. In practice, the MVCC problem can have a serious impact on the performance of the blockchain, as recent studies show that in realistic scenarios such as electronic health records, 40% of the transactions failed due to concurrency conflict [4]. Prior works seek to address or mitigate the MVCC problem by using a lockless approach to provide transaction isolation [12] or reordering transactions at the ordering phase to minimize the conflict rate when validating and committing transactions to the ledger [16] [15].

In our paper, we present a novel approach to solve the MVCC problem called *Early Detection of MVCC Conflicts* (EMVCC). We introduce an EMVCC detection mechanism that aims to reduce the number of conflicts between transactions which increases the overall system goodput. The advantage of our solution over existing works is that the MVCC conflict is detected at the first contact of the transaction with the blockchain network at the endorsement policy allowing to improve the network performances.

The contributions of our paper are as follows:

1) We provide the formulation of the problem of early detection of conflicting transactions at the endorsement phase, rather the traditional approach of handling MVCC conflicts at the validation phase (Section IV-B).
2) We propose a novel solution called Early Detection of MVCC Conflicts using local caching at endorsing nodes (Section IV-C). We provide a theoretical analysis of our solution to calculate the expected effectiveness of the solution depending on parameters such as the endorsement policy, number of organisations, etc. (Section IV-D).
3) We present three reference implementations of our solution: SyncMap, Lock-Free, and Mutex Lock, their main difference is the data structure used for information storing (Section IV-E).
4) We evaluate our solution and compare it to the baseline and we did a sensitivity analysis to study the impact of different network and load parameters on the performances (Section V).

The paper now continues with Section II that gives an overview of the Hyperledger Fabric. This is followed by Section III where we describe prior works related to the MVCC problem.

II. BACKGROUND

In this section, we give an overview of Hyperledger Fabric Components, we detail the transaction flow and we present the endorsement policies and Multi-Version Concurrency Control Read-Write Conflict (MVCC).

A. Hyperledger Fabric Components

Hyperledger Fabric is a permissioned distributed ledger system specialized for business applications. Its modular architecture makes blockchain solutions confidential, resilient, and flexible. Hyperledger Fabric has multiple fundamental components such as the client and the peers which are the endorsing peers and the committing peers. In this paper, we will focus on these components as they are the only components affected by our solutions.

1) Clients: Clients are applications or software that operate on behalf of a user to submit transactions on the Fabric network through the Hyperledger Fabric SDK.

2) Peers: Peers are a fundamental component of the network because they keep records of the network's state and a copy of the ledger. Each peer on the network belongs to an organization that may have one or more nodes. The administrator can create, start, stop or reconfigure the peer. To be part of the network, the node should join the shared channel between all the participants. There are two types of peers, endorsing peers and validating/committing peers. In a typical scenario, every Fabric peer typically fulfill both roles of endorsing and validating.

Endorsing peers: this type of node hosts the smart contract executed to simulate transactions and endorse them using the peer signature. When installing the chaincode (smart contract), the administrator specifies which peers will endorse the transactions by defining the endorsement policy rules.

Validating/Committing peers: they are responsible for the Validation System Chaincode (VSCC) used to validate the endorsement policies and the MultiVersion Concurrency Control (MVCC) validation to ensure that the version of the keys read during the endorsement phase is still the same in the ledger to guarantee the deterministic execution of the transaction. Then the committing peers add blocks to the shared ledger and update the blockchain world state.

B. Transaction Flow

Figure 1 shows the basic transaction flow in Hyperledger Fabric. It consists of five steps:

- **Step 1**: A client who wants to make a transaction sends a transaction proposal containing the chaincode function and arguments that the client wants to invoke, to the endorsing peer according to the chaincode endorsement policies (see the next Section II-C).
- **Step 2**: An endorser receives the transaction proposal, executes the chaincode, and generates the read/write sets which are the sets containing the values and versions of the keys which are read or written while executing the transaction,

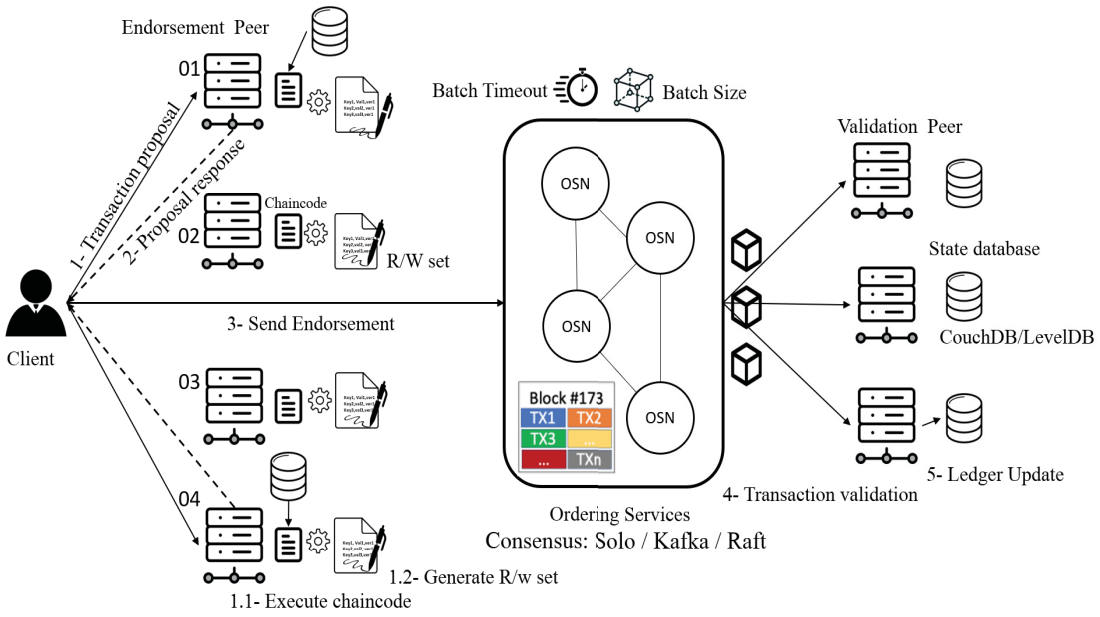


Fig. 1: Transaction Flow

then the peer creates the proposal response, signs it and sends the response to the client.

- **Step 3**: Once the client collects the required number of endorsements as defined in the endorsement policies, he sends the transaction to the ordering services, this operation contains details about the initial proposal as well as all peer endorsements and read/write sets. The ordering services order transactions received from clients into a block considering the block BatchSize and block BatchTimeout, then the block is created and delivered to the validating peer.
- **Step 4**: Upon receiving the block, the committing peer iterates over all the transactions within the block to perform the syntax validation of each transaction, the VSCC validation, and the MVCC validation (see the next Section II-D).
- **Step 5**: If a transaction is marked as valid after passing the three checks, it will be added to the new block and the ledger is updated by applying the transaction write sets, else invalid transactions will be rejected and the clients have to resubmit them.

C. Endorsements Policies

Hyperledger Fabric allows developers to set policies at the chaincode level. The endorsement policies are rules which specify which peers can agree on the transaction execution before it is added to the ledger. In general, endorsement policies are configured while installing the chaincode and can be modified only during a chaincode upgrade. Once the client creates the transaction, it sends a transaction proposal to all endorsing peers that satisfy the endorsement policy [11] and waits for the proposal responses. When the client receives enough responses and signatures to satisfy the endorsement policy of the chaincode, it can submit the transaction with the endorsement signatures to the ordering services. Endorsement policies can be defined as follow:

- `AND('Org1.member','Org2.member')`: the client needs one signature from each organization to be able to submit a transaction
- `OR('Org1.member','Org2.member')`: the client needs one signature from either one of the two organization to be able to submit a transaction
- `OR('Org1.member',AND('Org2.member','Org3.member'))`: the client needs either one signature from a member of the Organization1 or one signature from a member of the Organization2 and one signature from a member of the Organization3
- `Out-Of(2,'Org1.member','Org2.member','Org3.member')`: At least two of the three organization must endorse the transaction to be valid. It is equivalent to `OR(AND('Org1.member','Org2.member'),AND('Org1.member','Org3.member'),AND('Org2.member','Org3.member'))`

In the scope of this paper, we study the first two examples (AND, OR) for our solution, however types 3 and 4 have been studied but are not included in this paper due to lack of space. Furthermore, we assume that each client will select endorsing peers randomly

if given the choice in accordance to the policy.

D. Multi-Version Concurrency Control Read-Write Conflict (MVCC)

Hyperledger Fabric uses a Multi-version concurrency control system to ensure the consistency of the ledger. This mechanism validates that the versions of keys read at the endorsement time of the transaction are still the same at the validation phase [10]. This process guarantees that there are no reads of old values that have been changed by another concurrent transaction. During the time period between the endorsement and the validation phase, if another transaction has updated the version of the keys listed in the read set, the transaction will fail at the MVCC validation.

The multi-version concurrency control read-write conflict is a problem that occurs when two clients try to update and read the same key at the same time. Figure 2 shows an example of this conflict. Let's assume that *user1* submits *TX1*, and at the same time *user2* submits *TX2*, the two transactions will read and update the same value *Value1* of *Key1*. After endorsing and ordering simultaneously the transactions, *TX1* will be validated and its write set will be applied to the ledger resulting in the modification of the value and version associated with *Key1* to *value1* and *version1*. However, when *TX2* underwent the MVCC validation, it fails because the version of *Key1* in the read set of the transaction is not the same in the ledger. Therefore, the MVCC validation detects the inconsistency between the ledger and the endorsement result and return an MVCC error.

III. RELATED WORKS

Hyperledger Fabric is a relatively new system that is already experiencing some major architectural improvements. The majority of related work aims to improve the throughput and minimize the latency of the network, but there is a lack of effective solutions to deal with the MVCC problem and in most cases, the transaction conflict factor is not considered in the evaluation results. In this section, we review recent research on techniques to improve the Hyperledger Fabric performances. We will review this works along three categories: works that optimize endorsement phase, works that improve the ordering phase, and works that enhance the validation phase.

A. Endorsement Phase Optimization

The work done by Kwon et al. [9] is aiming to improve the read transaction processing by distinguishing between reading and writing transactions during the endorsement process. As a result, the transaction endorsement latency is reduced by 60% compared to the traditional fabric network. This approach is complementary to our own solution since early detection can still be applied to further reduce the latency of failed transactions.

B. Ordering Phase Optimization

For the ordering phase, István et al. [7] propose the elimination of the concept of blocks in favor of processing transactions in streaming to decrease the batching overhead. Also, the authors implement an FPGA-based consensus for the ordering service that decreases the commit latency below a millisecond by cutting latency in half compared to the Raft-based ordering service.

The authors of Fast Fabric [6] redesign the fabric ordering service to work with only the transaction IDs by Separating the transaction header from the payload to decide the transaction order only with the transaction's IDs which makes transaction processing in the ordering services faster which increase the throughput.

Sharma et al. [16] propose Fabric++ a solution that aims to reduce the MVCC failure rate by reordering transactions at the ordering phase using a conflict graph to abort the transaction that cannot be serialized.

Ruan et al. [15] designed an optimized extension of Fabric++ that can handle both inter-block and intra-block MVCC read conflicts.

Our solutions detect the MVCC conflict at the endorsement phase, however, Fabric++ and FabricSharp aim to reorder transactions in the ordering phase to avoid the maximum number of conflicts.

C. Validation Phase Optimization

Multiple articles propose the parallel execution of the validation process (syntax verification, endorsement policy verification, MVCC validation) to accelerate the block validation [5, 6, 7, 8, 17].

FabricCRDT focuses on automatically merging the conflicting transactions using CRDT techniques without rejecting them [14]. István et al. [7] introduce a disk writes batching mechanism by using a local batcher that accumulates the ledger update operations until reaching a batch size or batch time out to write to the database.

As an extension to batching writes, Thakkar et al. [17] propose bulk read operations during the MVCC validation to make a single API call per block to the database.

Finally, Meir et al. [12] presented a lock-free solution for providing transaction isolation, this approach allows the removal of the shared lock while ensuring transaction isolation.

The proposed improvements of the validation phase are based on the parallelization of validation processes or the optimization of reading and writing operations

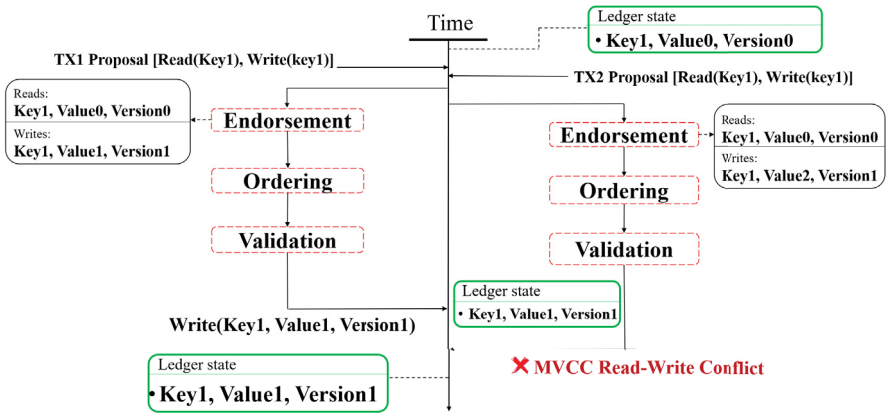


Fig. 2: MVCC Read-Write Conflict Example

on the databases. All of these works offer useful insights into various techniques that can improve the Hyperledger Fabric performance and represent a complementary improvement to our proposed solution.

IV. EMVCC Detection & Solution

In this section, we formulate the problem of MVCC detection at the endorsement phase. We then describe our proposed solutions called early MVCC detection, and present the different possible caching data structures. Also, we present a theoretical analysis of different endorsement policies.

A. System Model

In this section, we will use the following model: given an Hyperledger Fabric[1] network containing N organizations where each organization has M peers and M_i is the number of peers for organisation i. The system uses $Nb_{Tx/Block}$ as the number of transactions per block. The ordering services of the system can use any consensus protocol available in Hyperledger Fabric (e.g., Raft, Kafka, Solo). The chaincode executed by the application generates $\%conflict$ as the conflict rate representing the percentage of transactions which abort due to an MVCC conflict error. Intuitively speaking, a chaincode where transactions read and write to a few shared keys will have a higher conflict rate than that of a chaincode where transactions are independently working on disjoint keys. In order to store chain state, the system can be implemented using any of the available databases compatible with Hyperledger Fabric (CouchDB, LevelDB, etc.).

B. Problem Formulation

The problem we are solving is detecting MVCC conflict by filtering the transactions at the *endorsement phase* based on the endorsed transactions history of each endorsing peer. This is in contrast to the current problem resolved by Fabric, which is that of detecting MVCC conflicts at the *validation phase*, which occurs later in the transaction flow.

We start by defining some basic terms that we will use later:

- **Probability of non-detection:** This is a metric to evaluate the performance of our solution. A perfect solution would have a $\mathcal{P}(ND) = 0$. It represents the probability that our proposed solution does not detect a conflicting transaction at the endorsement phase, even though it has a conflict with another transaction, such that the MVCC conflict is only detected later at the validation phase.
- **False positives:** We call a false positive a transaction which is declared wrongly declared invalid using our solution, but would have been successfully confirmed using the standard approach. This can occur if our solution is overly aggressive in detecting conflicts and will prematurely abort transactions which have a chance to be successful, which would force the client to retry the entire endorsement process with a new transaction. The false positive rate can be calculated as follows:

$$FP = \frac{Nb_{Tx/Block} \times \%conflict^2 \times \mathcal{P}(ND) \times (1-\mathcal{P}(ND))}{\%conflict \times \mathcal{P}(ND) - \%conflicts + 1} \quad (1)$$

  where $Nb_{Tx/Block}$is the number of transactions per block, $\mathcal{P}(ND)$ is the probability of non-detection and $\%conflict$ is the conflict rate
- **False negatives:** It is the opposite error where a transaction is declared valid at the EMVCC validation stage while it has conflicts with another transaction resulting in its failure later on. Ideally, the number of false negatives should be equal to zero.

[1]Our solution has been tested with Fabric version 2.0 and higher.

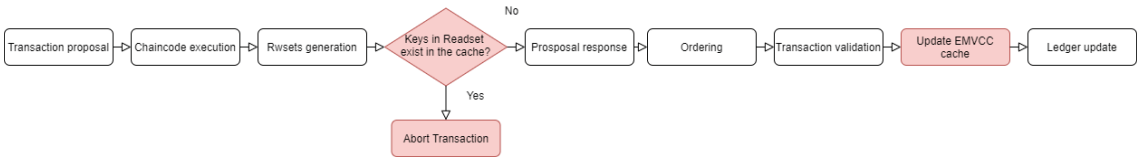


Fig. 3: EMVCC Detection Mechanism

$$FN = Transaction\_rate \times \%conflict \times \mathcal{P}(ND) \quad (2)$$

- **Goodput**[20] is the rate of successful transactions that the network can write into the blockchain. In contrast, the throughput of the network is the rate of total transactions which passed through the ordering phase, including transactions which will be aborted due to MVCC conflicts. Ideally the goodput should be equal to the throughput. In a standard Fabric implementation, the goodput is directly impacted by the MVCC conflict rate, since any aborted transaction is still part of a proposed block. Our solution can prevent such conflicts from being reflected inside the blockchain by detecting them prior to the ordering phase and thus improve the goodput. We can calculate it as follows:

$$goodput = \frac{Nb\_valid\_transactions}{Nb\_transactions} \times Throughput \quad (3)$$

C. Proposed Solution

In the current implementation of Fabric, a conflicting transaction passes through the network and eventually fails, thereby consuming unnecessary resources. In our proposed solution, we add a layer to filter transactions at the endorsement phase to abort any detected conflicting transactions. We call our approach Early Multi-Version Concurrency Control (EMVCC).

As described in Figure 3, our EMVCC solution operates after Fabric executes the chaincode and generates the read/write sets, and before an endorsement response is sent back to the client. Our EMVCC solution will compare the read set to a list of pending transactions stored in the peer's local cache. This cache is populated with transactions currently in progress this peer has previously endorsed which have not yet been confirmed (or aborted).

If a transaction reads a key that is stored in the cache, the transaction will be aborted since an instance of early MVCC conflict was detected. If not, the transaction is stored in the local cache and an endorsement is sent as usual back to the client. Once the transaction is validated or rejected at the validating phase, the cache is updated by removing the keys of the write set of this transaction.

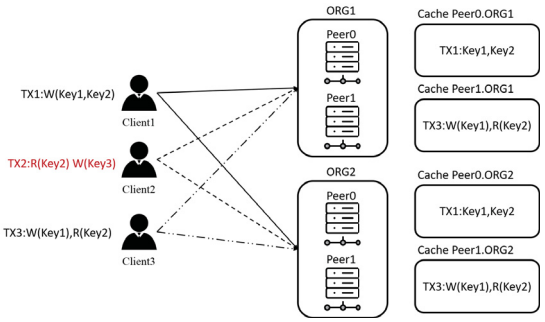


Fig. 4: EMVCC Detection Example

Note that the usual MVCC phase is still performed during the validation phase, since our EMVCC does not detect conflicts perfectly (see Section IV-D). This allows the regular validation phase to catch any MVCC conflicts that are not detected through our EMVCC solution, thereby maintaining correctness of the EOV execution flow.

Figure 4 shows an example of how EMVCC detection works. Suppose that we have a Fabric network composed of two organizations, each organization having two peers. The chaincode endorsement policy is AND (Org1,Org2). Suppose that *client1* submits *TX1* which writes on *Key1* and *Key2*. This transaction is endorsed by *Peer0.ORG1* and *Peer0.ORG2*. Now suppose we have a transaction *TX2* which reads *Key2* and writes on *Key3*. If this transaction is endorsed by *Peer0.ORG1* or *Peer0.ORG2*, it will be aborted at the endorsement phase because *TX1* is in the EMVCC cache of *Peer0.ORG1* or *Peer0.ORG2*. Suppose another transaction *TX3* arrives which writes on *Key1* and reads *Key2*. If it is endorsed by *Peer1.ORG1* and *Peer1.ORG2*, it will not be detected by the EMVCC detection mechanism because these peers does not have any conflict with this keys in their EMVCC cache. However, it will still fail later on at the MVCC validation phase.

D. Theoretical Analysis of Non-Detection

Since our approach is lightweight in nature and only uses local caching without resorting to any synchronization, it is possible for a conflicting transaction to pass the EMVCC phase undetected. This can happen if the subset of peers chosen to endorse the current transaction have not previously endorsed a pending transaction which conflicts with it. Thus, the endorse-

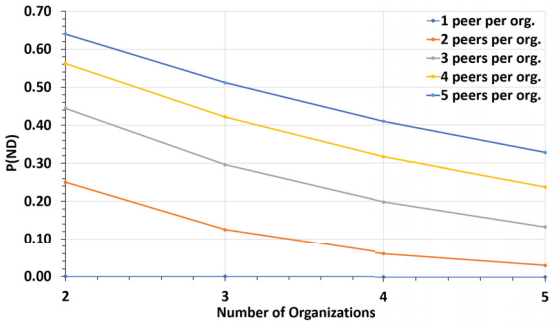


Fig. 5: Probability of Non-Detection using AND

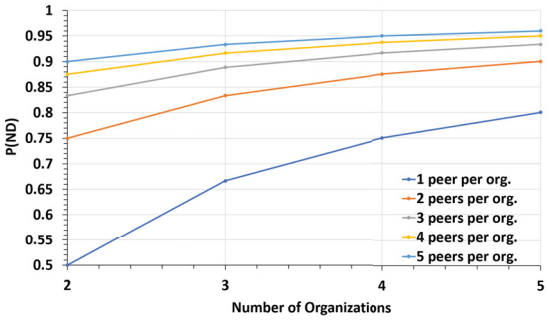


Fig. 6: Probability of Non-Detection using OR

ment policy, as well as the number of peers and the number of organisations, have a direct impact on the probability of non-detection of our proposed EMVCC solution.

In order to assess the performance of our solution, we present a analytical model which calculates the expected probability of non-detection for our solution using the AND and OR endorsement policies. These theoretical results will be compared to the real results in Section V to validate the correctness of our implementation. For this analysis, we assume the selection of peers (and organisations) is done uniformly at random when a client is presented with a choice of multiple peers (or organisations) to satisfy the endorsement policy, as it is currently implemented in Hyperledger Fabric [11]. Due to the lack of space, we defer the full details of our mathematical formulas derived here to our extended report.

1) Non-Detection in the AND Endorsement Policy: Using the AND endorsement policy, the probability of non-detection can be calculated as follows:

$$\mathcal{P}(ND) = \prod_{n=1}^{N} \frac{(M_i - 1)}{M_i} \tag{4}$$

where N is the number of organizations and M_i is the number of peers per organization

Figure 5 plots the theoretical evolution of the probability of non-detection of the EMVCC solution for various organizations and peer numbers using AND endorsement policy. We observe that the probability of non-detection decreases when the number of organizations increases, however, it increases when the number of peers per organization increases.

2) Non-Detection in the OR Endorsement Policy: We also analyze the evolution of the probability of non-detection using the OR endorsement policy, it can be calculated using the following formula:

$$\mathcal{P}(ND) = \begin{cases} \frac{NM-1}{NM} & \text{if } M_i{=}M_{i+1} \\ \frac{1}{N}\sum_{i=1}^{N}\frac{1}{M_i} & \text{else} \end{cases} \tag{5}$$

where N is the number of organizations and M_i is the number of peers per organization

Figure 6 illustrates the theoretical evolution of the probability of non-detection for different organizations and peers' numbers using OR endorsement policy. We identify that the probability of non-detection increases when the number of organizations or the number of peers increases due to the lack of synchronization between nodes caches.

E. Choice of Caching Data Structure

For the local cache maintained by each peer, we propose three different techniques in our reference implementation:

MutexLock: In this implementation, we are using a lock-based data structure to ensure cache consistency while using a multithreaded system [1]. For example, when an endorser reads or writes on the cache, it will be locked and the validator cannot update this entry in the cache. The main disadvantage of this approach is that the locking system adds a delay to the read and write operations since some threads have to wait until a lock is released.

LockFree: With this approach, more than one thread can access the cache concurrently. The read and writes operations are stored in a buffer then once the buffer threshold is reached, the batch is applied to the cache which guarantees that the majority of threads make progress at each step [1]. This technique is useful to decrease the number of times a lock is acquired or released.

SyncMap: Sync Map is an optimized and safe data structure, which is part of the sync package since Golang 1.9. It uses a dirty map to write new values or updates allowing the read operation to be done without a lock. However, the Mutex locking is essential if numerous concurrent threads (*goroutines*) are writing concurrently in the dirty map and the read-only map is updated by a batch of operations from the dirty map.

V. BENCHMARKING RESULTS

In this section, we evaluate the effectiveness of our proposed EMVCC solution using three different cache implementations: Mutex Lock, LockFree, and SyncMap. We first compare our three different implementations against the standard Fabric implementation as a baseline. We then perform a sensitivity analysis to study the influence of certain system parameters and workload characteristics on the system performance. Table I shows the default used parameters for our experiment. We use 500 transactions per block as the block size and a block time of 2 seconds which are the ordering parameters that lead to the highest throughput from our experiential testing (see Section V-D). Furthermore, we use a conflict rate of 40% because in realistic scenarios it represents the percentage of failed transactions due to concurrency conflict [4]. We chose the Raft consensus over the the two alternative ordering services because the solo mode is recommended only for development and testing purposes [18] and Kafka is deprecated in versions 2.x of Fabric. We also chose 3 ordering services nodes to satisfy the trade-off between the cost of the network and its performance.

A. Test Environment Characteristic

Our cluster consists of six E2 virtual machines hosted on the Google Cloud Platform. Two machines serve as two peers and three machines running the ordering services with a raft consensus algorithm. The other virtual machine serves as clients by running the Hyperledger Caliper [19] benchmarking tool. Each virtual machine has 8 vCPUs and 32 GB of RAM. All the virtual machines are running Ubuntu 16.04 as an operating system. The Fabric peers are set up to use our modified version of Hyperledger 2.3 images and CouchDB as the state database. For the chaincode, we use the Fabcar chaincode which allows us to create a car on the blockchain and modify its owner.

B. Performances Metrics

The main metrics for our benchmarking are:

1) **Goodput**: the effective throughput of committed transactions written to the blockchain excluding the aborted ones (Section IV-B).
2) **Latency**: the time between the initial request by the client of the transaction and its final commit to the ledger.
3) **EMVVC vs. MVCC rate**: the percentage of rejected transactions due to an EMVCC error versus the percentage of rejected transactions due to an MVCC error.
4) **TD EMVCC and TD MVCC**: the time-to-detect is the total time duration between the submission of the transaction by the client and its rejection by an MVCC or EMVCC error, respectively.

TABLE I: Default Evaluation Parameters

Parameters	Values
Block size	500 TXs
Block time	2 seconds
Number of organizations	2
Number of peers per organization	2
Ordering Consensus	Raft (3 Orderers)
MVCC Conflict rate	40%
Endorsement Policy	AND(Org1,Org2)

C. Comparison With The Baseline Solution

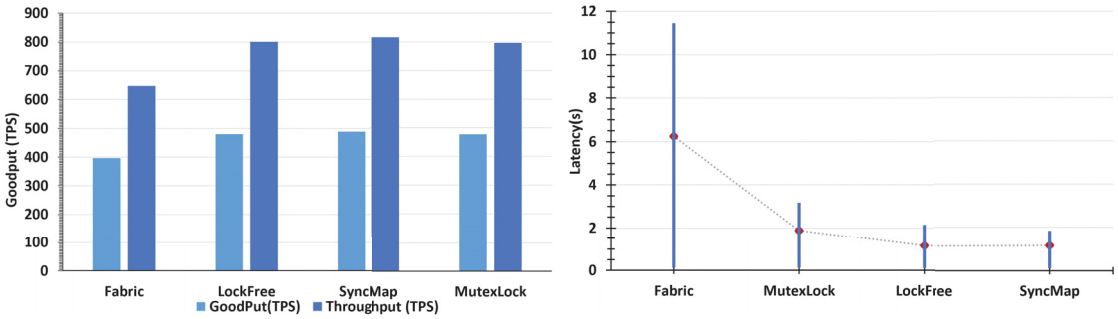
In this section, we compare our proposed three solutions with version 2.3 of Hyperledger Fabric. The throughput and transaction latency are averaged over ten runs.

Figure 7a plots the average throughput for three implementations and the Hyperledger Fabric. The three proposed solutions are better than the Fabric baseline, the best solution being SyncMap with a 23.2% of goodput improvement compared to Fabric. For Mutex Lock and LockFree, the percentage of improvement is 20.4% and 21.2%, respectively. Also, Figure 7b plots the average latency for the three implementations and the baseline. Similarly, the three implementations reduce latency compared to the baseline: for SyncMap and LockFree, the latency is reduced by 80%, and for MutexLock by 69%. This is due to the early detection of the conflicting transaction at the endorsement instead of going until the validation phase to be aborted. We thus conclude that SyncMap is the best data structure for our early MVCC cache implementation because it is the best for read operations when the the peer has multiple vcPUs.

D. Sensitivity Analysis

In this section, we analyse the impact of various network and workload parameters on the performance metrics such as the conflict rate, endorsement policies, and compute resources per peer, etc.

1) Impact of Conflict Rate: The conflict rate is the most important parameter that can show us the utility of our proposed solution. In Figure 8a, we plot the average goodput for Fabric and our different solutions over different values of conflict rates. As expected, with an increase in the conflict rate, the goodput decreases because we are increasing the number of failed transactions until we reach a 100% conflict rate where the goodput becomes zero. Also, we can see that the three proposed solutions perform better than Fabric for different values of the conflict rate. SyncMap is the best one, since it is improving the goodput by 10% when the conflict is around 20% and its impact is more important by exceeding 20% to reach an improvement of 23% at a conflict rate of 40%. The relative performance of this solution increases over the baseline as the conflict rate increases for an application.



(a) Throughput Comparison With The Baseline Solution

(b) Latency Comparison With The Baseline Solution

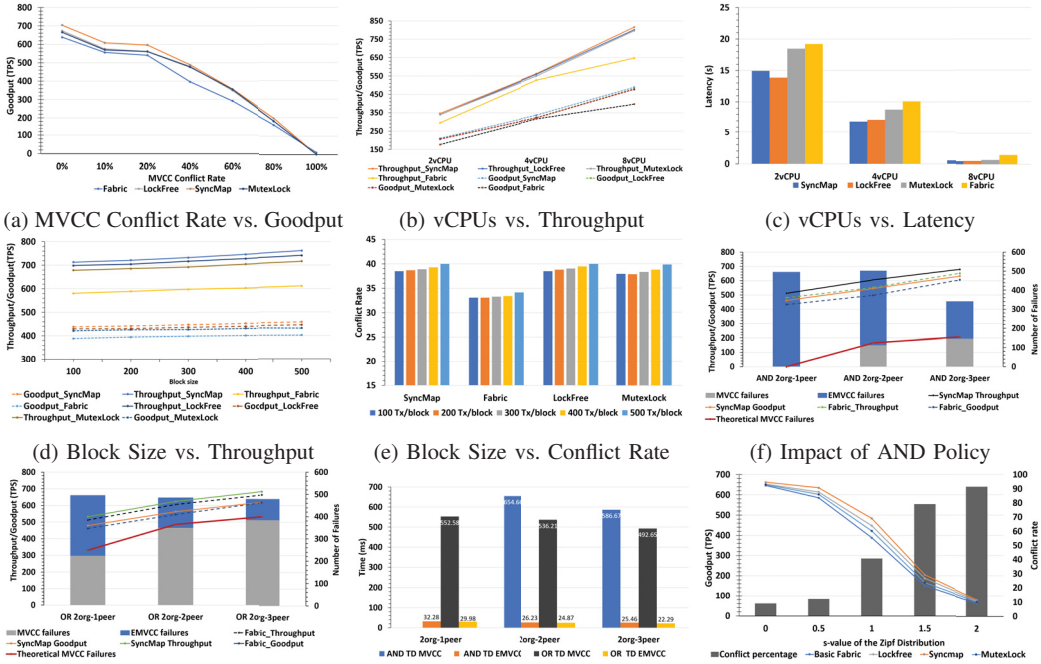
Fig. 7: Comparison With The Baseline Solution

2) Impact of Compute Resources per Peer: In this part, we will analyze the impact of adding or removing vCPUs to each peer. We vary the number of CPU from 2 vCPUs to 8 vCPUs. Figure 8b plots the average throughput and goodput and Figure 8c plots the average latency for our three implementations and the Hyperledger Fabric baseline for various vCPU numbers per peer. With an increase in the number of vCPUs per peer, the goodput increases and the latency decreases. Our three solutions are better than the baseline at varying numbers of vCPUs. However, we observe that with 8 vCPUs, there is a significant improvement of 23% for goodput and 65% for latency between SyncMap and the baseline. Thus, we conclude that our solutions are able to better leverage additional computational resources than the Fabric baseline.

3) Impact of Block Size: We analyse the impact of the block size by varing the block size from 100 transactions per block to 500 transactions per block. Figure 8d plots the average throughput and goodput for Fabric and our three implementations over various block sizes. Increasing the block size increases the throughput and goodput for all solutions because the use of larger blocks will cause less overhead and fewer network communications. The SyncMap method is offering an average improvement of 12% compared to Fabric over different block sizes which confirms that our proposed solutions are scalable with different block sizes. Figure 8e plots the conflict rate for Fabric and our three methods over various block sizes. We can observe that with an increase in the block size the conflict rate increases: this is due to the increase of the number of intra-block conflicts when the block contains a higher number of transactions. One other observation that can be made is that the conflict rate for Fabric is less than our proposed implementation: this is because there are some false positives and negatives caused by the early MVCC detector.

4) Impact of Endorsement Policies and Network Topology: For this experiment, we use different network topologies by varying the number of peers per organization from one peer to three peers. Figures 8f and 8g plot the average throughput and goodput, as well as the percentage of transaction failure, caused by EMVCC and MVCC validation for different network topologies using AND (Org1, Org2) and OR (Org1, Org2) endorsement policies for the Fabric baseline and SyncMap solutions. We observe that the type of endorsement policy impacts the network performances. As shown in the two figures, the throughput for the OR endorsement policy is higher than the throughput using the AND policy because using OR the transaction needs less peer endorsement than the AND. For the percentage of non-detection, these experimental results confirm the theoretical results presented in Section IV. For example, with AND Policy and using a network with two organizations each one has two peers, we have 22% of conflicting transactions that were not detected at the EMVCC and were aborted at the MVCC validation (false negative). We can also see that increasing the number of peers increases throughput and goodput for both endorsement policies as the organizations have more resources to process transactions. We can also observe that the increase in the number of peers per organization increases the rate of false positives. The improvement when the AND policy is used is more important compared to Fabric. However, with OR policy, the number of false positives increases which impacts the solution's performance. We can conclude that with an increase in the number of endorsers our solution becomes more efficient.

Figure 8h plots the average time to detect EMVCC/MVCC over different network topologies using OR and AND endorsement policies. The duration of the transaction execution is reduced by 95% between aborting a conflicting transaction at the EMVCC phase and processing it until reaching the MVCC validation phase to be rejected that is why our solutions reduce significantly the transaction latency. We note that when the number of peers per organization increases, the time to detect MVCC decreases due to the availability of computational resources.



(a) MVCC Conflict Rate vs. Goodput (b) vCPUs vs. Throughput (c) vCPUs vs. Latency

(d) Block Size vs. Throughput (e) Block Size vs. Conflict Rate (f) Impact of AND Policy

(g) Impact of OR Policy (h) Time To Detect EMVCC/MVCC (i) Conflict Skewness

Fig. 8: Sensitivity Analysis Results

5) Impact of Chaincode Implementation: To simulate another chaincode behavior, we use the Zipf distribution, which allows us to choose the keys used to simulate transactions by varying the parameter s of the distribution from 0 to 2. By increasing s, we are increasing the preference to use certain same keys which increase the conflict rate. Figure 8i plots the average goodput for the Fabric, SyncMap, LockFree, and MutexLock solutions over different values of the parameter s of the Zipf distribution. We can see that when s increases the conflict rate increases causing the goodput to decrease. As expected, when the s parameter increases our solutions perform better than Fabric. For example, SyncMap improves the goodput by 24% for s equal to 1 and 35% for s equal 1.5.

E. Summary and Discussions of Findings

Our three proposed solutions perform better than the Fabric baseline. SyncMap is the best data structure to implement the cache for keys storing. The improvement rate compared to Fabric is mostly determined by the used endorsement policies, the network design, and the chaincode implementation (conflict rate). In realistic scenarios, where 40% of the transactions failed to MVCC conflict, the SyncMap solution improves the throughput by 23% and reduces the latency by 80%.

When using our solution, we recommend developers to use the AND endorsement policy with a maximum number of organizations and the minimum number of peers per organization if the business logic allows that. However, when they have to use the OR endorsement policy, we recommend that they use the minimum number of organizations and number of peers per organization. The block size should be adjusted carefully to minimize the inter-block and intra-block transactions conflict in order to maximize the goodput. For the chaincode implementation, it is important to write a chaincode that ensures a conflict rate lower than 40%.

VI. CONCLUSIONS

In this work, we propose a mechanism to improve Hyperledger Fabric performances aiming to early abort transactions that have no chance to be validated and committed to the ledger using different caching techniques. In an experimental evaluation, we compare SyncMap, LockFree, and MutexLock to the basic version of Fabric. We also did a sensitivity analysis by varying configurable parameters such as block size, endorsement policies, and resource allocation. We show that the proposed solutions outperform Fabric and SyncMap which is the best-proposed method that improves the throughput by up to 20% and reduces the latency by up to 80% compared to Fabric.

REFERENCES

[1] Andrei Alexandrescu. “Lock-Free Data Structures”. In: (2007), p. 7.

[2] Elli Androulaki et al. “Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains”. In: *Proceedings of the 13th EuroSys Conference, EuroSys 2018* 2018-Janua (2018). DOI: 10.1145/3190508.3190538.

[3] Vitalik Buterin. “A next-generation smart contract and decentralized application platform”. In: *Etherum* January (2014), pp. 1–36. URL: http://buyxpr.com/build/pdfs/EthereumWhitePaper.pdf.

[4] Jeeta Ann Chacko, Ruben Mayer, and Hans-Arno Jacobsen. “Why Do My Blockchain Transactions Fail? A Study of Hyperledger Fabric (Extended version)*”. In: 4 (2021). URL: http://arxiv.org/abs/2103.04681.

[5] Christian Gorenflo, Lukasz Golab, and Srinivasan Keshav. “XOX Fabric: A hybrid approach to transaction execution”. In: *PhD Seminar , Systems and Networking* (2019). URL: http://arxiv.org/abs/1906.11229.

[6] Christian Gorenflo et al. “FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second”. In: *ICBC 2019 - IEEE International Conference on Blockchain and Cryptocurrency* (2019), pp. 455–463. DOI: 10.1109/BLOC.2019.8751452.

[7] Zsolt István, Alessandro Sorniotti, and Marko Vukolić. “StreamChain: Do blockchains need blocks?” In: *SERIAL 2018 - Proceedings of the 2018 Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers* (2018), pp. 1–6. DOI: 10.1145/3284764.3284765.

[8] Haris Javaid, Chengchen Hu, and Gordon Brebner. “Optimizing Validation Phase of Hyperledger Fabric”. In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2019), pp. 269–275. DOI: 10.1109/mascots.2019.00038.

[9] Minsu Kwon and Heonchang Yu. “Performance Improvement of Ordering and Endorsement Phase in Hyperledger Fabric”. In: *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)* (2019), pp. 428–432.

[10] Per-åke Larson et al. “High-Performance Concurrency Control Mechanisms for Main-Memory Databases”. In: (2012), pp. 298–309.

[11] Yacov Manevich, Artem Barger, and Yoav Tock. “Service Discovery for Hyperledger Fabric”. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems* (May 2018), pp. 226–229. DOI: 10.1145/3210284.3219766. URL: http://arxiv.org/abs/1805.02105%20https://dl.acm.org/doi/10.1145/3210284.3219766.

[12] Hagar Meir et al. “Lockless transaction isolation in hyperledger fabric”. In: *Proceedings - 2019 2nd IEEE International Conference on Blockchain, Blockchain 2019* (2019), pp. 59–66. DOI: 10.1109/Blockchain.2019.00017.

[13] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. In: *Journal for General Philosophy of Science* 39.1 (2008), pp. 53–67. ISSN: 09254560. DOI: 10.1007/s10838-008-9062-0.

[14] Pezhman Nasirifard and Ruben Mayer. “FabricCRDT : A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains”. In: *the 20th International Middleware Conference* (2019), pp. 110–122.

[15] Pingcheng Ruan et al. “A Transactional Perspective on Execute-order-validate Blockchains”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2020), pp. 543–557. ISSN: 07308078. DOI: 10.1145/3318464.3389693.

[16] Ankur Sharma et al. “Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric CCS CONCEPTS • Information systems → Distributed database trans-actions; ACM Reference Format”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* 18 (2019), pp. 105–122. URL: https://doi.org/10.1145/3299869.3319883.

[17] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. “Performance benchmarking and optimizing hyperledger fabric blockchain platform”. In: *Proceedings - 26th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2018* (2018), pp. 264–276. DOI: 10.1109/MASCOTS.2018.00034.

[18] Canhui Wang and Xiaowen Chu. “Performance characterization and bottleneck analysis of hyperledger fabric”. In: *Proceedings - International Conference on Distributed Computing Systems* 2020-Novem (2020), pp. 1281–1286. DOI: 10.1109/ICDCS47774.2020.00165.

[19] Clemens Wickboldt. “Benchmarking a Blockchain-based Certification Storage System”. In: (2019).

[20] Ruozhou Yu et al. “CoinExpress: A fast payment routing mechanism in blockchain-based payment channel networks”. In: *Proceedings - International Conference on Computer Communications and Networks, ICCCN* 2018-July

(2018). ISSN: 10952055. DOI: 10.1109/ICCCN.2018.8487351.

[21] Zibin Zheng et al. “An overview on smart contracts: Challenges, advances and platforms”. In: *Future Generation Computer Systems* 105 (2020), pp. 475–491. ISSN: 0167739X. DOI: 10.1016/j.future.2019.12.019.

ANNEXE II

CHAINCODE UTILISÉ POUR EMVCC

```
package main

import (
        "encoding/json"
        "fmt"
        "strconv"

        "github.com/hyperledger/fabric-contract-api-go/contractapi"
)

// SmartContract provides functions for managing a car
type SmartContract struct {
        contractapi.Contract
}

// Car describes basic details of what makes up a car
type Car struct {
        Make   string `json:"make"`
        Model  string `json:"model"`
        Colour string `json:"colour"`
        Owner  string `json:"owner"`
}

// QueryResult structure used for handling result of query
type QueryResult struct {
        Key    string `json:"Key"`
        Record *Car
}

// InitLedger adds a base set of cars to the ledger
func (s *SmartContract) InitLedger(ctx contractapi.TransactionContextInterface) error {
        cars := []Car{
                Car{Make: "Toyota", Model: "Prius", Colour: "blue", Owner: "Tomoko"},
                Car{Make: "Ford", Model: "Mustang", Colour: "red", Owner: "Brad"},
                Car{Make: "Hyundai", Model: "Tucson", Colour: "green", Owner: "Jin_Soo"},
                Car{Make: "Volkswagen", Model: "Passat", Colour: "yellow", Owner: "Max"},
                Car{Make: "Tesla", Model: "S", Colour: "black", Owner: "Adriana"},
                Car{Make: "Peugeot", Model: "205", Colour: "purple", Owner: "Michel"},
                Car{Make: "Chery", Model: "S22L", Colour: "white", Owner: "Aarav"},
                Car{Make: "Fiat", Model: "Punto", Colour: "violet", Owner: "Pari"},
```

```
                Car{Make: "Tata", Model: "Nano", Colour: "indigo", Owner: "Valeria"},
                Car{Make: "Holden", Model: "Barina", Colour: "brown", Owner: "Shotaro"},
        }

        for i, car := range cars {
                carAsBytes, _ := json.Marshal(car)
                err := ctx.GetStub().PutState("CAR"+strconv.Itoa(i), carAsBytes)

                if err != nil {
                        return fmt.Errorf("Failed␣to␣put␣to␣world␣state.␣%s", err.Error())
                }
        }

        return nil
}

// CreateCar adds a new car to the world state with given details
func (s *SmartContract) CreateCar(ctx contractapi.TransactionContextInterface, carNumber string,
make string, model string, colour string, owner string) error {
        car := Car{
                Make:   make,
                Model:  model,
                Colour: colour,
                Owner:  owner,
        }

        carAsBytes, _ := json.Marshal(car)

        return ctx.GetStub().PutState(carNumber, carAsBytes)
}

// QueryCar returns the car stored in the world state with given id
func (s *SmartContract) QueryCar(ctx contractapi.TransactionContextInterface, carNumber string)
(*Car, error) {
        carAsBytes, err := ctx.GetStub().GetState(carNumber)

        if err != nil {
                return nil, fmt.Errorf("Failed␣to␣read␣from␣world␣state.␣%s", err.Error())
        }

        if carAsBytes == nil {
                return nil, fmt.Errorf("%s␣does␣not␣exist", carNumber)
        }
```

```
        car := new(Car)
        _ = json.Unmarshal(carAsBytes, car)

        return car, nil
}

// QueryAllCars returns all cars found in world state
func (s *SmartContract) QueryAllCars(ctx contractapi.TransactionContextInterface)
([]QueryResult, error) {
        startKey := "CAR0"
        endKey := "CAR99"

        resultsIterator, err := ctx.GetStub().GetStateByRange(startKey, endKey)

        if err != nil {
                return nil, err
        }
        defer resultsIterator.Close()

        results := []QueryResult{}

        for resultsIterator.HasNext() {
                queryResponse, err := resultsIterator.Next()

                if err != nil {
                        return nil, err
                }

                car := new(Car)
                _ = json.Unmarshal(queryResponse.Value, car)

                queryResult := QueryResult{Key: queryResponse.Key, Record: car}
                results = append(results, queryResult)
        }

        return results, nil
}

// ChangeCarOwner updates the owner field of car with given id in world state
func (s *SmartContract) ChangeCarOwner(ctx contractapi.TransactionContextInterface, carNumber string,
newOwner string) error {
        car, err := s.QueryCar(ctx, carNumber)

        if err != nil {
```

```
                return err
        }

        car.Owner = newOwner

        carAsBytes, _ := json.Marshal(car)

        return ctx.GetStub().PutState(carNumber, carAsBytes)
}

func main() {

        chaincode, err := contractapi.NewChaincode(new(SmartContract))

        if err != nil {
                fmt.Printf("Error␣create␣fabcar␣chaincode:␣%s", err.Error())
                return
        }

        if err := chaincode.Start(); err != nil {
                fmt.Printf("Error␣starting␣fabcar␣chaincode:␣%s", err.Error())
        }
}
```

ANNEXE III

CHAINCODE UTILISÉ POUR MDBC

```
package main

import (
        "encoding/json"
        "errors"
        "fmt"
        "strconv"

        "github.com/hyperledger/fabric-chaincode-go/pkg/cid"
        "github.com/hyperledger/fabric-chaincode-go/shim"
        pb "github.com/hyperledger/fabric-protos-go/peer"
        "github.com/hyperledger/fabric/common/flogging"
)

const (
        BalancePrefix   = `BALANCE`
        AllowancePrefix = `APPROVE`
)

var (
        ErrNotEnoughFunds                    = errors.New(`not␣enough␣funds`)
        ErrForbiddenToTransferToSameAccount = errors.New(`forbidden␣to␣transfer␣to␣same␣account`)
        ErrSpenderNotHaveAllowance           = errors.New(`spender␣not␣have␣allowance␣for␣amount`)
)

type Balance struct {
        Amount float64 `json:"Amount"`
}

// SimpleChaincode example simple Chaincode implementation
type SimpleChaincode struct {
}

func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
        return shim.Success(nil)
}

var logger = flogging.MustGetLogger("fabcar_cc")

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
```

```
        function, args := stub.GetFunctionAndParameters()
        fmt.Println("invoke is running " + function)

        // Handle different functions
        if function == "Init" { //create a new marble
                return t.Init(stub)
        }
        if function == "invokeTransfer" { //transfer money
                return t.invokeTransfer(stub, args)
        }
        if function == "BalanceOf" { //get balance
                return t.BalanceOf(stub, args)
        }
        if function == "SetBalance" { //set balance
                return t.SetBalance(stub, args)
        }
        if function == "GetUserIdentity" { //get user identity atrribute
                return t.GetUserIdentity(stub, args)
        }
        fmt.Println("invoke did not find func: " + function) //error
        return shim.Error("Received unknown function invocation")
}

// Init initializes the chaincode
func (t *SimpleChaincode) invokeTransfer(stub shim.ChaincodeStubInterface, args []string) pb.Response {

        type AmountTransientInput struct {
                Amount string `json:"amount"`
        }

        transMap, err := stub.GetTransient()
        if err != nil {
                return shim.Error("Error getting transient: " + err.Error())
        }

        AmountAsBytes, ok := transMap["amount"]
        if !ok {
                return shim.Error("Amount must be a key in the transient map")
        }

        if len(AmountAsBytes) == 0 {
                return shim.Error("Amount value in the transient map must be a non-empty JSON string")
        }
```

```
var AmountInput AmountTransientInput
err = json.Unmarshal(AmountAsBytes, &AmountInput)
if err != nil {
        return shim.Error("Failed␣to␣decode␣JSON␣" + err.Error())
}

// transfer target
ReceiverMspId := args[0]
ReceiverCertId := args[1]

//transfer amount
Amount, _ := strconv.ParseFloat(AmountInput.Amount, 32)

SenderMspId := args[2]
SenderCertId := args[3]

// Disallow to transfer Receiverken to same account
if SenderMspId == ReceiverMspId && SenderCertId == ReceiverCertId {
        return shim.Error("forbidden␣to␣transfer␣to␣same␣account")
}

ReceiverBalance, err := getBalance(stub, ReceiverMspId, ReceiverCertId)
if err != nil {
        return shim.Error("Can␣not␣get␣Receiver␣balance" + err.Error())
        //return shim.Error(collectionName)
}

SenderBalance, err := getBalance(stub, SenderMspId, SenderCertId)
if err != nil {
        return shim.Error("Can␣not␣get␣Sender␣balance")
}
out, _ := json.Marshal(SenderBalance)
// Check the funds sufficiency
if SenderBalance-Amount < 0 {

        return shim.Error(string(out))
}

// Update payer and Receiver balance
if err = setBalance(stub, SenderMspId, SenderCertId, SenderBalance-Amount); err != nil {
        return shim.Error("Can␣not␣update␣Sender␣balance")
}

if err = setBalance(stub, ReceiverMspId, ReceiverCertId, ReceiverBalance+Amount);
```

```
        err != nil {
                return shim.Error("Can␣not␣update␣receipient␣balance")
        }

        return shim.Success([]byte(out))
}

func (t *SimpleChaincode) BalanceOf(stub shim.ChaincodeStubInterface, args []string) pb.Response {

        MSPID := args[0]
        CertID := args[1]

        InvokerMspId, _ := cid.GetMSPID(stub)
        InvokerCertId, _ := cid.GetID(stub)

        if MSPID != InvokerMspId || CertID != InvokerCertId {
                return shim.Error("You␣are␣not␣allowed␣to␣get␣this␣balance")
        }

        Balance, err := getBalance(stub, MSPID, CertID)
        if err != nil {
                return shim.Error(err.Error())
        }
        BalanceAsBytes, _ := json.Marshal(Balance)

        return shim.Success(BalanceAsBytes)

}

func (t *SimpleChaincode) SetBalance(stub shim.ChaincodeStubInterface, args []string) pb.Response {

        MSPID := args[0]
        CertID := args[1]
        Amount2, _ := strconv.ParseFloat(args[2], 32)
        //if err != nil {
        //      return shim.Error("Parse failed" + err.Error())
        //}
        err2 := setBalance(stub, MSPID, CertID, Amount2)
        out, _ := json.Marshal(Amount2)
        if err2 != nil {
                return shim.Error("Error:␣Balance␣not␣setted:␣" + err2.Error())
        }
        return shim.Success([]byte(out))
```

```
}

func (t *SimpleChaincode) GetUserIdentity(stub shim.ChaincodeStubInterface, args []string)
pb.Response {

        MSPID, _ := cid.GetMSPID(stub)
        CertID, _ := cid.GetID(stub)

        identity := fmt.Sprintf(MSPID, CertID)

        return shim.Success([]byte(identity))

}

// Internal Function
// setBalance puts balance value to state
func balanceKey(ownerMspId, ownerCertId string) string {
        s := fmt.Sprintf(BalancePrefix, ownerMspId, ownerCertId)
        return s
}

func getBalance(stub shim.ChaincodeStubInterface, mspId, certId string) (float64, error) {
        var balance Balance

        collectionName := getCollectionName(stub, mspId)
        BalanceAsBytes, err := stub.GetPrivateData(collectionName, balanceKey(mspId, certId))
        if err != nil {
                return -1, err
        }
        json.Unmarshal(BalanceAsBytes, &balance)
        return balance.Amount, nil
}

// setBalance puts balance value to state
func setBalance(stub shim.ChaincodeStubInterface, mspId, certId string, amount float64) error {
        balance := Balance{
                Amount: amount,
        }
        BalanceAsBytes, _ := json.Marshal(balance)

        collectionName := getCollectionName(stub, mspId)
        return stub.PutPrivateData(collectionName, balanceKey(mspId, certId), BalanceAsBytes)
```

```
}

//get collection name
func getCollectionName(stub shim.ChaincodeStubInterface, MSPID string) string {
        collection_name := ""
        if MSPID == "Org1MSP" {
                collection_name = "BOC_collection"
        } else if MSPID == "Org2MSP" {
                collection_name = "org2_collection"
        } else if MSPID == "Org3MSP" {
                collection_name = "org3_collection"
        } else if MSPID == "Org4MSP" {
                collection_name = "org4_collection"
        } else if MSPID == "Org5MSP" {
                collection_name = "org5_collection"
        } else {
                fmt.Printf("No␣Organization␣collection")
        }
        return collection_name
}
func main() {
        err := shim.Start(new(SimpleChaincode))
        if err != nil {
                fmt.Printf("Error␣starting␣Simple␣chaincode:␣%s", err)
        }
}
```

LISTE DE RÉFÉRENCES

Albert, E., Correas, J., Gordillo, P., Román-Díez, G. & Rubio, A. (2020). GASOL : Gas Analysis and Optimization for Ethereum Smart Contracts. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12079 LNCS(April), 118–125. doi : 10.1007/978-3-030-45237-7_7.

Alonso, S. L. N., Jorge-Vazquez, J. & Forradellas, R. F. R. (2021). Central banks digital currency : Detection of optimal countries for the implementation of a CBDC and the implication for payment industry open innovation. *Journal of Open Innovation : Technology, Market, and Complexity*, 7(1), 1–23. doi : 10.3390/joitmc7010072.

Androulaki, E., Barger, A., Bortnikov, V., Muralidharan, S., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Murthy, C., Ferris, C., Laventman, G., Manevich, Y., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S. W. & Yellick, J. (2018). Hyperledger Fabric : A Distributed Operating System for Permissioned Blockchains. *Proceedings of the 13th EuroSys Conference, EuroSys 2018*, 2018-Janua. doi : 10.1145/3190508.3190538.

Baliga, A., Solanki, N., Verekar, S., Pednekar, A., Kamat, P. & Chatterjee, S. (2018). Performance characterization of hyperledger fabric. *Proceedings - 2018 Crypto Valley Conference on Blockchain Technology, CVCBT 2018*, 65–74. doi : 10.1109/CVCBT.2018.00013.

Buterin, V. (2014). A next-generation smart contract and decentralized application platform. *Etherum*, 1–36. Repéré à http://buyxpr.com/build/pdfs/EthereumWhitePaper.pdf.

Chacko, J. A., Mayer, R. & Jacobsen, H.-A. (2021). Why Do My Blockchain Transactions Fail ? A Study of Hyperledger Fabric (Extended version). Repéré à http://arxiv.org/abs/2103.04681.

China_Bank. (2021). Progress of Research Development of E-CNY in China. Repéré à http://www.pbc.gov.cn/en/3688110/3688172/4157443/4293696/2021071614584691871.pdf.

Cunha, P. R., Melo, P. & Sebastião, H. (2021). From bitcoin to central bank digital currencies : Making sense of the digital money revolution. *Future Internet*, 13(7), 1–19. doi : 10.3390/fi13070165.

Dalessandro, L., Dice, D., Scott, M., Shavit, N. & Spear, M. (2010). Transactional Mutex Locks. *Euro-Par 2010 - Parallel Processing*, pp. 2–13.

Danezis, G. & Meiklejohn, S. (2015). Centrally Banked Cryptocurrencies. *Computing Research Repository (CoRR)*. Repéré à http://arxiv.org/abs/1505.06895.

Ding, D., Li, K., Jia, L., Li, Z., Li, J. & Sun, Y. (2019). Privacy protection for blockchains with account and multi-asset model. *China Communications*, 16(6), 69–79. doi : 10.23919/j.cc.2019.06.006.

Eliazar, I. (2016). Zipf law : an extreme perspective. *Journal of Physics A : Mathematical and Theoretical*, 49(15), 15LT01. doi : 10.1088/1751-8113/49/15/15lt01.

Eyal, I., Gencer, A. E., Sirer, E. G. & Van Renesse, R. (2016). Bitcoin-NG : A scalable blockchain protocol. *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016*, 45–59.

Gilbert, S. & Lynch, N. (2012). Perspectives on the CAP Theorem. *Computer*, 45(2), 30-36. doi : 10.1109/MC.2011.389.

Gorenflo, C., Golab, L. & Keshav, S. (2019a). XOX Fabric : A hybrid approach to transaction execution. *PhD Seminar, Systems and Networking*. Repéré à http://arxiv.org/abs/1906.11229.

Gorenflo, C., Lee, S., Golab, L. & Keshav, S. (2019b). FastFabric : Scaling Hyperledger Fabric to 20,000 Transactions per Second. *ICBC 2019 - IEEE International Conference on Blockchain and Cryptocurrency*, 455–463. doi : 10.1002/nem.2099.

Han, R., Foutris, N. & Kotselidis, C. (2019a). Demystifying Crypto-Mining : Analysis and Optimizations of Memory-Hard PoW Algorithms. *Proceedings - 2019 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019*, 22–33. doi : 10.1109/ISPASS.2019.00011.

Han, X., Yuan, Y. & Wang, F. Y. (2019b). A Blockchain-based Framework for Central Bank Digital Currency. *Proceedings - IEEE International Conference on Service Operations and Logistics, and Informatics 2019 (SOLI 2019)*, 263–268. doi : 10.1109/SOLI48380.2019.8955032.

István, Z., Sorniotti, A. & Vukolić, M. (2018). StreamChain : Do blockchains need blocks ? *SERIAL 2018 - Proceedings of the 2018 Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 1–6. doi : 10.1145/3284764.3284765.

Javaid, H., Hu, C. & Brebner, G. (2019). Optimizing Validation Phase of Hyperledger Fabric. *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 269–275. doi : 10.1109/mascots.2019.00038.

Kaneko, Y., Osada, S., Azuchi, S., Okada, H. & Yamasaki, S. (2019). A Management Method of Interest-rate in UTXO Model. 1–6. doi : 10.1109/sitim.2019.8910267.

Kwon, M. & Yu, H. (2019). Performance Improvement of Ordering and Endorsement Phase in Hyperledger Fabric. *2019 Sixth International Conference on Internet of Things : Systems,*

Management and Security (IOTSMS). doi : 10.1109/IOTSMS48152.2019.8939202.

Larson, P.-, Blanas, S., Diaconu, C., Freedman, C., Patel, J. M. & Zwilling, M. (2012). High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proceedings of the VLDB Endowment*, 298–309. doi : 10.14778/2095686.2095689.

Li, X. & Li, Z. (2020). A performance measurement and optimization mechanism for blockchain mining pool system. *ACM International Conference Proceeding Series*, 27–33. doi : 10.1145/3446983.3446991.

Manevich, Y., Barger, A. & Tock, Y. (2018). Service Discovery for Hyperledger Fabric. *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, 226–229. doi : 10.1145/3210284.3219766.

Meir, H., Barger, A., Manevich, Y. & Tock, Y. (2019). Lockless transaction isolation in hyperledger fabric. *Proceedings - 2019 2nd IEEE International Conference on Blockchain, Blockchain 2019*, 59–66. doi : 10.1109/Blockchain.2019.00017.

Nakamoto, S. (2008). Bitcoin : A Peer-to-Peer Electronic Cash System. *Journal for General Philosophy of Science*, 39(1), 53–67. doi : 10.1007/s10838-008-9062-0.

Nasirifard, P. & Mayer, R. (2019). FabricCRDT : A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains. *the 20th International Middleware Conference*, 110–122. doi : 10.1145/3361525.3361540.

Naudts, E., Aerts, T., franken, l. & Pieterse, A. (2021). Governance in Systems Based on Distributed Ledger Technology (DLT) : A Comparative Study. *SSRN Electronic Journal*, (718). doi : 10.2139/ssrn.3876116.

Nguyen, T. S. L., Jourjon, G., Potop-Butucaru, M. & Thai, K. L. (2019). Impact of network delays on Hyperledger Fabric. *INFOCOM 2019 - IEEE Conference on Computer Communications Workshops, INFOCOM WKSHPS 2019*, 222–227. doi : 10.1109/INFCOMW.2019.8845168.

Pérez-Solà, C., Delgado-Segura, S., Navarro-Arribas, G. & Herrera-Joancomartí, J. (2019). Another coin bites the dust : An analysis of dust in UTXO-based cryptocurrencies. *Royal Society Open Science*, 6(1). doi : 10.1098/rsos.180817.

Qin, R., Yuan, Y. & Wang, F. Y. (2018). Research on the selection strategies of block-chain mining pools. *IEEE Transactions on Computational Social Systems*, 5(3), 748–757. doi : 10.1109/TCSS.2018.2861423.

Ruan, P., Loghin, D., Ta, Q. T., Zhang, M., Chen, G. & Ooi, B. C. (2020). A Transactional Perspective on Execute-order-validate Blockchains. *Proceedings of the ACM SIGMOD*

International Conference on Management of Data, 543–557. doi : 10.1145/3318464.3389693.

Sharma, A., Schuhknecht, F. M., Agrawal, D. & Dittrich, J. (2019). Blurring the Lines between Blockchains and Database Systems : the Case of Hyperledger Fabric ; ACM Reference Format. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 18, 105–122. Repéré à https://doi.org/10.1145/3299869.3319883.

Slepak, G. & Petrova, A. (2018). The DCS Theorem. *CoRR*, abs/1801.04335. Repéré à http://arxiv.org/abs/1801.04335.

Sun, H., Mao, H., Bai, X., Chen, Z., Hu, K. & Yu, W. (2018). Multi-blockchain model for central bank digital currency. *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings*, 2017-Decem, 360–367. doi : 10.1109/PDCAT.2017.00066.

Thakkar, P., Nathan, S. & Viswanathan, B. (2018). Performance benchmarking and optimizing hyperledger fabric blockchain platform. *Proceedings - 26th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2018*, 264–276. doi : 10.1109/MASCOTS.2018.00034.

Valois, J. D. *Lock-Free Linked Lists Compare and Swap*. (Thèse de doctorat, Rensselaer Polytechnic Institut).

Wang, C. & Chu, X. (2020). Performance characterization and bottleneck analysis of hyperledger fabric. *Proceedings - International Conference on Distributed Computing Systems*, 2020-Novem, 1281–1286. doi : 10.1109/ICDCS47774.2020.00165.

Wickboldt, C. (2019). Benchmarking a Blockchain-based Certification Storage System. doi : 10.13140/RG.2.2.32684.31360.

Wills, F. & Sanders, D. (2017). Applications of CBT. *Cognitive Behaviour Therapy : Foundations for Practice*, 272–295. doi : 10.4135/9781526435651.n11.

Yu, R., Xue, G., Kilari, V. T., Yang, D. & Tang, J. (2018). CoinExpress : A fast payment routing mechanism in blockchain-based payment channel networks. *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, 2018-July. doi : 10.1109/ICCCN.2018.8487351.

Zhang, J., Tian, R., Cao, Y., Yuan, X., Yu, Z., Yan, X. & Zhang, X. (2021). A Hybrid Model for Central Bank Digital Currency Based on Blockchain. *IEEE Access*, 9, 53589–53601. doi : 10.1109/ACCESS.2021.3071033.

Zheng, Z., Xie, S., Dai, H. N., Chen, W., Chen, X., Weng, J. & Imran, M. (2020). An overview on smart contracts : Challenges, advances and platforms. *Future Generation Computer Systems*,

105, 475–491. doi : 10.1016/j.future.2019.12.019.