World Wide Web (2021) 24:463–482
https://doi.org/10.1007/s11280-020-00851-6

Solutions for concurrency conflict problem on Hyperledger Fabric

Lu Xu[1] · Wei Chen[1] · Zhixu Li[1] · Jiajie Xu[1] · An Liu[1] · Lei Zhao[1]

Received: 31 March 2020 / Revised: 26 September 2020 / Accepted: 29 September 2020 /
Published online: 14 November 2020

Abstract
A Hyperledger Fabric is a popular permissioned blockchain platform and has great commercial application prospects. However, the limited transaction throughput of Hyperledger Fabric hampers its performance, especially when transactions with concurrency conflicts are initiated. In this paper, we focus on transactions with concurrency conflicts and propose solutions to optimize the performance of Hyperledger Fabric. Firstly, we propose a novel method LMLS to improve the Write-Write Conflict. This method introduces a lock mechanism in the transaction flow to enable some conflicting transactions to be marked at the beginning of the transaction process. And indexes are added to conflicting transactions to optimize the storage of the ledger. Secondly, we propose a cache-based method to improve the Read-Write Conflict. The cache is used to speed up reading data, and a cache log is added to Hyperledger Fabric to ensure the data consistency. Extensive experiments demonstrate that the proposed novel methods can significantly increase transaction throughput in the case of concurrency conflicts, and maintain high efficiency in transactions without concurrency conflicts.

Keywords Blockchain · Hyperledger fabric · Concurrency · Locking mechanism · Caching mechanism

1 Introduction

Blockchain technologies have become popular these years and can be applied to different domains. Unlike a common database system, a Blockchain is a distributed, shared ledger

This article belongs to the Topical Collection: *Special Issue on Web Information Systems Engineering 2019*
Guest Editors: Reynold Cheng, Nikos Mamoulis, and Xin Huang

✉ Wei Chen
robertchen@suda.edu.cn

✉ Lei Zhao
zhaol@suda.edu.cn

Extended author information available on the last page of the article.

system where the nodes do not fully trust each other. Each node holds the copy of the ledger which is represented as a chain of blocks, with each block being a sequence of transactions. With the characteristics of decentralization, distrust and tamper-proof, blockchain is adopted in a wide variety of industries. A number of blockchain platforms have been developed, including Bitcoin [13], Ethereum [7], Hyperledger Fabric [12] etc. Among them, Hyperledger Fabric is a representative blockchain platform and has attracted much attention due to the wide application range of it.

Hyperledger Fabric is a permissioned blockchain platform which is highly suitable for developing enterprise-class applications and has a modular design. In Hyperledger Fabric, the identity of each participant is known and authenticated cryptographically. Different from many blockchains whose nodes are peer-to-peer, nodes in Hyperledger Fabric are of different types. The nodes in Hyperledger Fabric contain Client, Peer and Orderer, and each of them performs individual duty in the transaction flow. A transaction is initiated by Client and sent to endorsing Peers. Endorsing Peers do endorsement and send response to Client, then Client broadcasts the transaction proposal and response to Orderer which orders them into blocks. The blocks containing some transactions are delivered to all Peers. At last, Peers update the ledger and the transaction flow finishes. In addition, Hyperledger Fabric has better scalability and security, and superior in performance [16] such as latency and throughput to other blockchain platforms. Hyperledger Fabric which our work focuses on is currently being used in many different applications such as Global Trade Digitization [23], SecureKey [19] and Everledger [8].

Hyperledger Fabric has received a lot of concerns, but has exposed many problems at the same time. The main problem is the performance of transaction processing, that is, blockchain system including Hyperledger Fabric can only handle a huge volume of transactions with a low throughput. Some papers analyze the performance of Hyperledger Fabric, Gupta et al. [10] [11] present two models to optimize the temporal query performance of Hyperledger Fabric. Thakkar et al. [22] study the impact of various configuration parameters on the performance of Hyperledger Fabric. Gorenflo et al. [9] improve the throughput of Hyperledger Fabric by reducing computation and I/O overhead during the transaction flow.

Although these studies have made great contributions, their proposed methods cannot be directly used to tackle the following task, i.e., multiple operations updating the same data in the ledger simultaneously. This is because approaches developed in existing work can only conduct the operations having no conflicting transactions. Unfortunately, this problem, which is called concurrency conflicts, is ubiquitous in Hyperledger Fabric where the data is distributedly stored. We define the concurrency conflict in Hyperledger Fabric as multiple proposals reading or writing the same data in the ledger simultaneously.

We analyze the concurrency probelm on Hyperledger Fabric and find that when read and write operations are performed on the platform at the same time, the read operations are much faster than the write operations, so the read operations are submitted before the update of the ledger by the write requests. As a result, the value read by the read operation is actually a old value. In addition, when there are multiple transactions to modify the same data at the same time, since a transaction passes through multiple nodes and the transaction flow is relatively complicated, the system will only process one of the transactions and modify the value successfully. The remaining transactions will return a "MVCC READ CONFLICT" error and cannot be successfully updated, which leads to the inefficient processing of transactions in Hyperledger Fabric. Therefore, we summarize the concurrency problem on Hyperledger Fabric into two categories: Read-Write Concurrency Conflict and Write-Write Concurrency Conflict. Among them, Read-Write Concurrency Conflict means that the read operations and the write operations are submitted at the same time for the same data, which leads to the submission of wrong data. Write-Write Concurrency Conflict means that there are multiple transactions were initiated at the same time and required to modify the same data in the ledger, causing some transactions to fail to update the ledger.

To address above mentioned problem, we propose a novel method LMLS and a Cache-based method. Since Read-Write Concurrency Conflict includes Write-Write Concurrency Conflict, we first propose a LMLS method [24] for Write-Write Concurrency Conflict. Firstly, a locking mechanism is proposed to discovery conflicting transactions at the beginning of the transaction flow. For example, there are two transactions that are transferred to the same account at the same time. Since the previous transaction first updated the account data, the conflict of data inconsistency occurred in the latter transaction, which caused the transfer to fail. If there are multiple times of the above transactions, the processing efficiency will be low. The locking mechanism can prevent some conflicting transactions from occupying resources of the nodes. We use redis [18] to implement the locking mechanism which mainly contains locking and unlocking. When a transaction request is initiated, it is first checked to ensure if its corresponding key is locked, thereby determining whether the transaction is a conflicting transaction. Moreover, a listener is used to control the lock and unlock operations. Secondly, based on the locking mechanism, database indexes corresponding to conflicting transactions are changed and temporally stored to improve processing efficiency. In Hyperledger Fabric, the data is stored as a key-value pair $\langle k, v\rangle$. We transform the index of the data corresponding to the conflict transaction from k to (k, d), where d is a unique identifier of a transaction and (k, d) is the composite key generated by k and d. This allows conflicting transactions who share the same key not to fail. Then, based on the LMLS method, we propose a cache-based method to improve the system for Read-Write Concurrency Conflict. This method adds a cache mechanism to the entire transaction flow so that when a read operation is initiated, it can first try to read the cached data. This can greatly reduce the time used for the read operation. In addition, in order to ensure the consistency of the cache and the ledger data , we add a cached log to prevent cache deletion failures that cause the cache and ledger data to be out of sync. Combined with the LMLS method mentioned above, it can effectively improve the efficiency of Read-Write Concurrency Conflict on Hyperledger Fabric. That is to say, based on LMLS and Cached-based methods, we can address concurrency conflicts on Hyperledger Fabric. To sum up, the contributions of this paper are as follows.

- To the best of our knowledge, we are the first to improve the performance of Hyperledger Fabric in transaction processing by considering concurrency conflicts.
- To tackle the issue of concurrency conflicts, firstly, we design a novel method LMLS which contains Locking Mechanism and Ledger Storage to improve Write-Write Concurrency Conflict. Then, we propose a Cache-based method to improve Read-Write Concurrency Conflict.
- The experimental results show that our method can significantly increase transaction throughput in the case of concurrency conflicts and maintain high efficiency in transactions without concurrency conflicts.

The rest of the paper is organized as follows: We present the related work in Section 2 and formulate the problem in Section 3. Section 4 gives a brief introduction of Hyperledger Fabric architecture. In Section 5 we propose LMLS method to improve the performance of Hyperledger Fabric with Write-Write Concurrency Conflict. In Section 5 we propose Cache-based method to improve the performance of Hyperledger Fabric with Read-Write

Concurrency Conflict. In Section 7, experiments are conducted to validate the effectiveness of the proposed methods. Finally, we conclude this paper in Section 8.

2 Related work

Efficient handling of concurrency conflicts is a hot research topic in distributed database, and conflicting transactions are also existing in Hyperledger Fabric which is a distributed system. Hyperledger Fabric is a recent system that is still undergoing rapid development. Hence, there is relatively little work on the performance analysis of the system or suggestions for architectural improvements. Next, we will introduce the recent work related to this research.

Analyzing blockchain performance. Blockchain performance analysis is an emerging area. Recently the BLOCKBENCH system [6] benchmarked the popular blockchain implementations - Hyperledger Fabric, Ethereum and Parity [15] against a set of database workloads. Similar efforts include - benchmarking Hyperledger Fabric and Ethereum against transactional workloads [16]. They find that Hyperledger Fabric outperforms Ethereum in all metrics. Our paper focuses on improve the performance of Hyperledger Fabric.

Analyzing Hyperledger Fabric performance. Some studies have also looked at performance studies of Hyperledger Fabric, and analyzed the performance from multiple perspectives. For example, Nasir et al. [14] compare the performance of Hyperledger Fabric 0.6 and 1.0 which find that the 1.0 version outperforms the 0.6 version. Baliga et al. [2] show that application-level parameters such as the read-write set size of the transaction and chaincode as well as event payload sizes significantly impact transaction latency.

Optimizing transaction processing performance. Many studies have proposed the optimization of the performance for processing transactions in Hyperledger Fabric [25]. In recent work, Thakkar et al. [22] study the impact of various configuration parameters on the performance of Hyperledger Fabric. They identify some major performance bottlenecks and provide some optimizations such as MSP cache, parallel VSCC validation. Gupta et al. [10] [11] present two models to optimize the temporal query performance of Hyperledger Fabric. Gorenflo et al. [9] improve the throughput of Hyperledger Fabric by reducing computation and I/O overhead during the transaction flow. Sharma et al. [20] study the use of database techniques to reorder transaction to remove serialization conflicts and abort transactions which have no chance to commit early to improve the performance of Hyperledger Fabric.

Optimizing other aspects of performance. In addition, some papers have optimized the performance of other aspects of Hyperledger Fabric, i.e., channel, oderer component. As known to all, Hyperledger Fabric's orderer component can be a bottleneck so Sousa et al. [21] study the use of the well-known BFT-SMART [3] implementation as a part of Hyperledger Fabric to improve it. Androulaki et al. [1] study the use of channels for scaling Fabric. However, this work does not present a performance evaluation to quantitatively establish the benefits from their approach. Raman et al. [17] study the use of lossy compression to reduce the communication cost of sharing state between Fabric endorsers and committers. However, their approach is only applicable to scenarios which are insensitive to lossy compression, which is not the general case for blockchain-based applications.

However, only few studies have looked at the issues concurrency conflicts on blockchain. Thus, to improve the performance of Hyperledger Fabric, we focuses on concurrency conflicts of transactions on this platform.

3 Problem definition

3.1 Concurrency conflicts problem in Hyperledger Fabric

Although Hyperledger Fabric has a higher transaction throughput than other permissioned blockchain systems and some papers have studied its transaction performance, they almost assume that multiple requests do not modify the same data in the ledger at the same time. For the Hyperledger Fabric blockchain platform, its essence as a distributed database also has the concurrency problems. On Hyperledger Fabric, every transaction that is initiated will pass through Client, Peer, Orderer, etc., each transaction will be processed asynchronously, and the process will continuously perform identity verification, key verification, etc. Work to ensure the security of the data, and different from traditional databases, the data being writen to the ledger needs to be verified by multiple nodes for consistency.The overall process is more complicated. The speed of data being writen to the ledger and the speed of data being written to traditional databases are very different. Simply treating the blockchain as an ordinary database to do concurrency control processing, we need to find a suitable method based on the characteristics of the blockchain.

We analyze the concurrency probelm on Hyperledger Fabric and find that when read and write operations are performed on the platform at the same time, the read operations are much faster than the write operations, so the read operations are submitted before the update of the ledger by the write requests. As a result, the value read by the read operation is actually a old value. In additon, when multiple requests want to modify the same data simultaneously, Hyperledger Fabric will process one of the requests and successfully modify the value, and the rest will return “MVCC_READ_CONFLICT” errors, which cannot be successfully updated. In detail, according to the transaction flow of Hyperledger Fabric, both requests should be sent to Peers for endorsement, and the results of endorsement will be sent to Orderer. Orderer packages and sorts the transaction proposals and responses, then send them to all Peers for final validation. In the process of validation, Peers need to ensure that the current state of the ledger is consistent with the state of the ledger in which the transaction is generated. When multiple requests are initiated at the same time, one of the requests update the value of the data first, causing errors in the remaining requests when the requests verify consistency and returning failures. Such concurrency conflicts result in lower efficiency in processing transactions.

3.2 Instance for conflicting transactions

Specifically, as shown in Figure 1, there are three people Tom, John and Amy. In the initial state, the account balance of Tom, John and Amy is $100, $50, and $0. At a certain point, Tom and John simultaneously transfer $10 to Amy, that is, there are two requests to update Amy’s account balance at the same time. Here, they are initiated almost simultaneously, through endorsement by Peers, ordering by Orderers. Then, the two transactions are packed into the block and successively delivered to Peers for verification. It should be noted that the transactions in the block contain much information, one of them is the status of the ledger when the transaction is initiated (here, the status of the ledger is the initial state

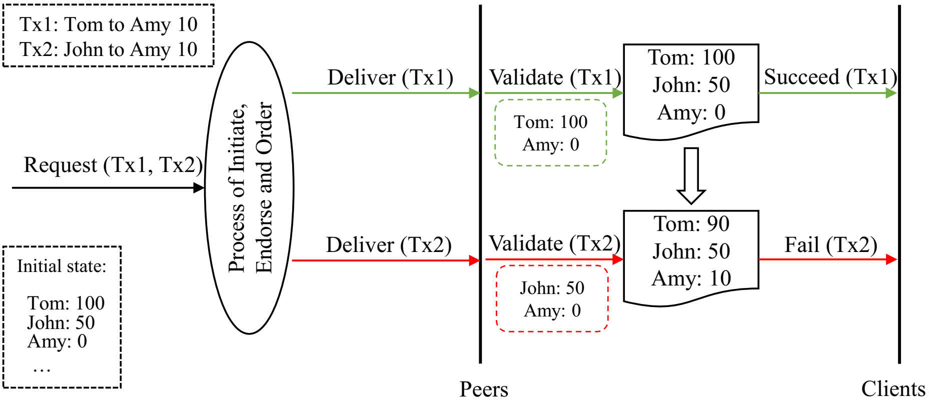

Figure 1 The instance for conflicting transactions (Tx1 represents Tom transfers $10 to Amy and Tx2 represents John transfers $10 to Amy)

shown in Figure 1). Without loss of generality, we assume that Tx1 arrives earlier, and Peers compare the local ledger with the initial state in Tx1(the values corresponding to Tom are both 100 and to Amy are both 0) finding that they are consistent. Therefore, the balance of Tom is successfully updated to $90 and the balance of Amy is successfully updated to $10. However, at this time, Tx2 is delivered to Peers, and repeating the above comparison, Peers find it is not consistent with the current value of the local ledger(the value corresponding to Amy in local ledger is 10 while the value in Tx2 is 0). Thus, the request of Tx2 is failed to update the ledger and it should be initiated again.

Problem formalization. On Hyperledger Fabric, multiple transactions are initiated simultaneously at the same time to modify the same data in the ledger, resulting in at least one transaction being successful and the remaining transactions failing. This situation is referred to as the Write-Write Concurrency Conflict of Hyperledger Fabric. On Hyperledger Fabric Blockchain, read transactions and write transactions that operate on the same data are submitted at the same time, resulting in the submission of erroneous data. This situation is called Write-Write Concurrency Conflict of Hyperledger Fabric.

4 Architecture of Hyperledger Fabric

4.1 Nodes in Hyperledger Fabric

Nodes are the communication entities of the blockchain. Different from many blockchains whose nodes are peer-to-peer, nodes in Hyperledger Fabric play different roles in the network. There are three types of nodes shown in Figure 2:

Client. A Client represents an entity operated by the end user. A Client submits transaction proposal to the Endorser Peer and broadcasts proposal and responses to Orderer.

Peer. A Peer is mainly responsible for reading and writing the ledger by executing chaincode. All Peers are committing peers (Committers) responsible for maintaining the state and the ledger. Peers can additionally take up a special role of an endorsing peer

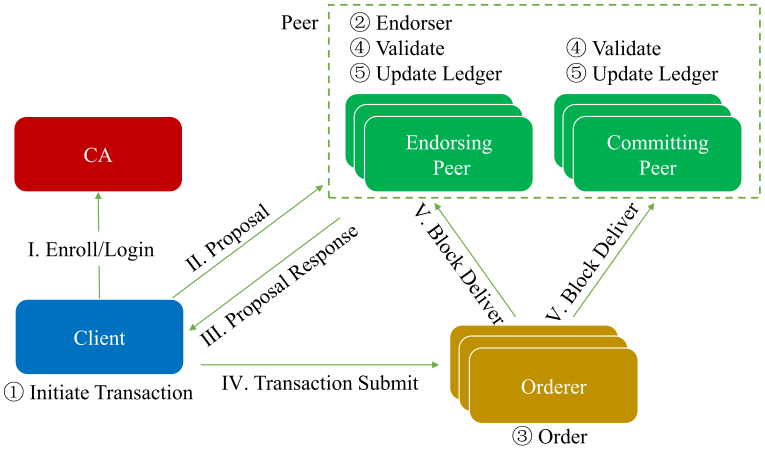

Figure 2 The transaction flow of Hyperledger Fabric

(Endorser). The endorsing peer is a dynamic role, and Peer is the endorsement node only when the application initiates a transaction endorsement request to it, otherwise it is a normal committing peer.

Orderer. A number of Orderers make up ordering service. Since the Hyperledger Fabric is a distributed system, a ledger is stored on each node. When each node wants to modify the state of the ledger, there must be a mechanism to ensure the consistency of all these operations,which is the orderer service. Orderers are responsible for ordering the unpackaged transactions into blocks.

4.2 Transaction flow

Figure 2 depicts the transaction flow which involves 5 steps. This flow assumes that the application user has registered and enrolled with the organization 's certificate authority (CA). The transaction flow is as follows:

1) Initiating Transaction. Client using Fabric SDK constructs a transaction proposal and sends the proposal which is signed with credentials to one or more endorsement Peers simultaneously.

2) Endorsement. First, the endorsing Peers verify the signature (using MSP). Second, the endorsing Peers take the transaction proposal arguments as inputs and execute the chaincode against the current state database to produce transaction results including a response, read set and write set. Third, the results, along with the endorsing Peer's signature and a YES/NO endorsement statement are passed back as a proposal response to Client. Client will collect enough proposal responses from Peers and verify if the result are same.

3) Ordering. Client broadcasts the transaction proposal and response within a transaction message to the Orderer. The Orderer orders them chronologically by channel, and creates blocks of transactions per channel.

4) Validation. The blocks containing some transactions are delivered to all Peers. Peers need to verify the signature by Orderer and need to do VSCC validation. A VSCC validation will check if the endorsement policy is satisfied, if not, the transaction will be marked invalid.

5) Ledger Updation. Each Peer appends the block to the local ledger, and for each valid transaction the write sets are committed to the state-db which stores the current state of all keys.

5 Proposed method LMLS

In order to solve the Write-Write Concurrency Conflict in Hyperledger Fabric, we propose the following novel method LMLS to optimize the transaction flow to increase efficiency. Firstly, a locking mechanism is proposed so that conflicting transactions can be discovered at the beginning of the transaction flow. Secondly, based on the lock mechanism, we add a database index for conflicting transactions and change the storage way of conflicting transactions, so that they can be temporarily stored in the database. The above methods can effectively improve the performance of Hyperledger Fabric with Write-Write Concurrency Conflict.

5.1 Locking mechanism

By analyzing the existing problems of Hyperledger Fabric, the main reason for the inefficiency is that invalid transactions (which ultimately failed to successfully update the ledger) are found to be invalid after almost completing the whole transaction flow. Therefore, we consider adding a locking mechanism at the beginning of the transaction process. The locking mechanism can prevent some of the conflicting transactions from occupying resources of the nodes, so that some invalid transactions can be found in the early stage of the transaction flow, thereby improving efficiency.

Implementation of the locking mechanism. In this paper, we use redis [18] to implement the locking mechanism. Redis is essentially a database of key-value types. Due to the advantages of redis in performance and concurrency, the use of redis scenarios is mostly a highly concurrent scenario. The idea of implementation is not complicated. In general, we can be divided into two steps: locking and unlocking. First introduce the process of locking, the distinguished name of a task in the request as a key to the redis. If there is a request with the same distinguished name arriving, try to insert it into redis. If it can be successfully inserted, return $True$, that is, it is successfully locked and will get a lock identifier. Otherwise, return $False$, that is, the other request with the same distinguished name is operating, and the lock fails. The process of unlocking is relatively simple. The lock identifier is passed as a parameter to check whether the lock exists. If it exists, the lock identifier can be deleted from the redis.

Listener. To determine when to unlock, we used a listener which can be used to know when the transaction was successfully written to the blockchain. Because of knowing that the transaction has been written to the block, the identifier can be unlocked. In this paper, we use Hyperledger Fabric officially provided listening interface ChannelEventHub [5]. Transaction processing in Hyperledger Fabric is a long operation. As a result the applications must design their handling of the transaction lifecycle in an asynchronous fashion. We mainly use registerTxEvent interface to listen the transaction flow. When a transaction is initiated, a transaction listener is registered and returns a specific sequence number as the identifier. When the transaction is written to the blockchain, it will be listened to by the listener, and the listener will call the function to unlock the lock identifier corresponding to the transaction.

5.2 Optimization of ledger storage

Although the lock mechanism can cause invalid transactions to be discovered earlier, users need to re-initiate these transactions which does not improve the user experience. When multiple conflicting transactions are initiated simultaneously, there will still be only one transaction that can be successfully updated to the blockchain ledger and the other transactions need to be initiated again. Therefore, based on the locking mechanism, we improve the storage of the blockchain ledger and transform the database indexes to avoid concurrency conflicts.

In Hyperledger Fabric, the data in ledger is stored in key-value pair. For a key k, the latest pair is called the current state of the key k which is stored in state-db, while all the pairs including the latest pair form the historical states of key k which is stored in history-db. Obviously, the collection of current states for all keys is termed as state-db, and the collection of historical states is termed as history-db. In this paper, all the changes transactions initiated are in the current state, so we only pay attention to state-db.

Usually, we modify the data in state-db by initiating a proposal. In this paper, we assume that each time a proposal is initiated, only one data in state-db is modified, that is, a transaction T generates a proposal P, which corresponds to a key-value pair $\langle k, v\rangle$ in state-db. If two transactions T_i and T_j are initiated at the same time, two proposals P_i and P_j will be generated, corresponding to the key-value pairs $\langle k_i, v_i\rangle$ and $\langle k_j, v_j\rangle$ in the state-db. If $k_i = k_j$, this is the case of concurrency conflicts. In order to effectively avoid conflicts and enable both proposals to be successfully executed, we transform the database indexes of state-db. Specifically, for conflicting transactions, we transformed $\langle k, v\rangle$ to $\langle (k, d), v\rangle$ where (k, d) is the composite key generated by k and d, and d is a transaction id for transaction T, which is a unique identifier that is randomly generated. For transactions T_i and T_j, without losing generality, we assume that T_i is processed before T_j, then we transform $\langle k_j, v_j\rangle$ to $\langle k'_j, v_j\rangle$ where k'_j represents the composite key (k_j, d_j). Thus, k_i and k'_j are not equal and both transactions T_i and T_j can update the ledger avoiding concurrency conflicts.

5.3 Steps of LMLS

Combining the ledger storage improvements with locking mechanism, the steps of LMLS are shown in Figure 3, which can be divided into the following steps.

I A user initiates a transaction, and Client pre-processes the transaction, including obtaining the key k of the data that the transaction wants to update. Client checks if k is locked. If it is, directly turn to III, otherwise, turn to II.

II Lock k and get a lock identifier l.

III Client opens the listener, generates the corresponding transaction proposal, and sends the proposal to Peers.

IV(i) If k obtains the corresponding lock identifier l, Peers generate the key-value pair $\langle k', v\rangle$ according to the transaction id, and endorse to simulate the execution of smart contracts.

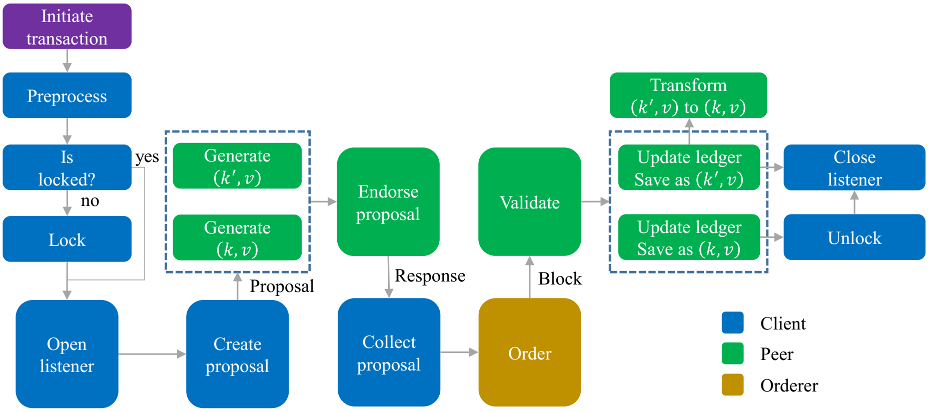

Figure 3 The complete transaction flow with LMLS

IV(ii) If k does not obtain l, Peers generate the key-value pair $\langle k, v\rangle$, and endorse to simulate the execution of smart contracts.

V Peers return the endorsement result to Client, and Client sends the proposal and result to Orderer which order and package them to new block. Orderer send the packaged block to Peers, and Peers perform the final verification.

VI(i)] If k obtains the corresponding lock identifier l, Peers save $\langle k', v\rangle$ into state-db to update ledger. Client listens to the operation and closes the listener.

VI(ii) If k does not obtain l, Peers save $\langle k, v\rangle$ into state-db to update the ledger. Client listens to the operation, then it unlocks the lock identifier l corresponding to k first and closes the listener.

VII After all the above steps are finished, $\langle k', v\rangle$ will merge with $\langle k, v\rangle$ by chaincode safely and the former will be deleted.

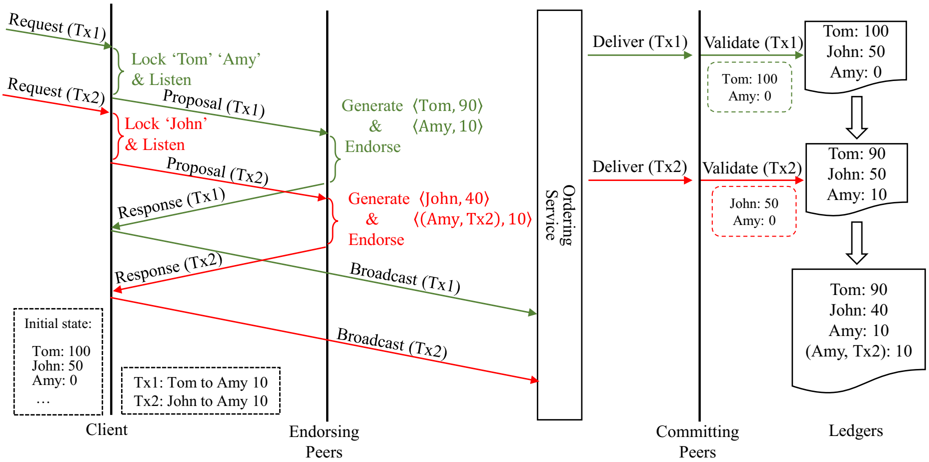

Figure 4 The example for LMLS to process conflicting transactions (Tx1 represents Tom transfers $10 to Amy and Tx2 represents John transfers $10 to Amy)

5.4 Example for LMLS

Continue the example in Section 3, we assume that Tx1 in Figure 4 arrives earlier, then Client locks two keys('Tom' and 'Amy') in Tx1 and starts listening. Subsequently, the request of Tx2 is initiated, at this time Client only locks the key 'John' , then starts listening. When the above two proposals are sent to Peers, Peers generate the corresponding key-value pair respectively and endorse them. The difference is that for Tx1, two key-value pairs $\langle Tom, 90\rangle$ and $\langle Amy, 10\rangle$ are generated, but for Tx2, a key-value pair $\langle John, 40\rangle$ and a composite index-key-value pair $\langle (Amy, Tx2), 10\rangle$ are generated. Then, the two transactions are ordered and delivered to Peers where validation need to be done. In this example, Peers first validate Tx1. They compare the local ledger with the initial state in Tx1(the values corresponding to Tom are both 100 and to Amy are both 0) finding that it is consistent. Therefore, the balance of Tom is successfully updated to $90 and the balance of Amy is successfully updated to $10. Next, Peers validate Tx2, since it has be known as a conflicting transaction in the previous process where the value corresponding to Amy is being operated by another request, a composite key-value pair $\langle (Amy, Tx2), 10\rangle$ will be added to the ledger instead of $\langle Amy, 20\rangle$. In addition, the balance of John will be successfully updated to $40. As shown in Figure 4, there are two indexes related to Amy in the final ledger where the sum of them is 20. When we request to query Amy's balance, it will return 20 instead of 10.

6 Proposed cache-based method

In order to solve the Read-Write Concurrency Conflict in Hyperledger Fabric, based on the LMLS method mentioned above, we propose the following Cache-based method. This method adds a cache mechanism to the entire transaction flow so that when a read operation is initiated, it can first try to read the cached data. This can greatly reduce the time used for the read operation. In addition, in order to ensure the consistency of the cache and the ledger data, we add a cached log to prevent cache deletion failures that cause the cache and ledger data to be out of sync.

6.1 Cache mechanism

By analyzing Read-Write Concurrency on Hyperledger Fabric, we find the main reason for the problem is that when read and write operations are performed on the platform at the same time, the read operations are much faster than the write operations, so the read operations are much faster than the write operations, so the read operations are submitted before the update of the ledger by the write requests. As a result, the value read by the read operation is actually an old value. To solve this problem, a Cache-based method is proposed, and its specific method is as shown in Figure 5.

First, for a write operation, after the user initiates a write operation, the Client node will first add a read-write lock to the data corresponding to the operation, and then update the ledger. After the update is completed, the listener will receive this information and then proceed. A check to check whether the operation is the last write operation of the current data.If it is, it means that there are no other operations to update the current data in the ledger, you can update the cache, and unlock the corresponding data after updating the cache, otherwise Unlock directly. At this point, the write operation is complete. Next, for the read operation, after the user initiates the read operation, the Client node will first check whether the corresponding data is added with a read-write lock. If so, it means that a write

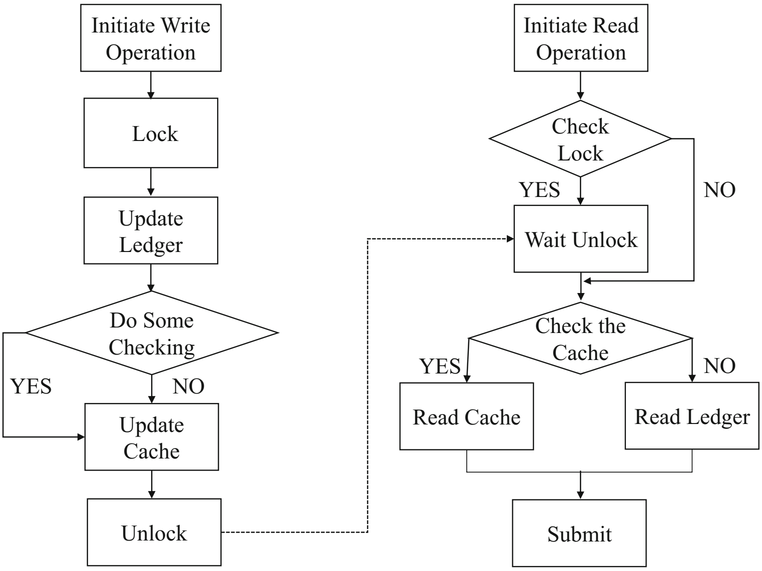

Figure 5 The cache flow

operation is updating the data at this time, so wait, otherwise start the check. Cache, if there is a cache, then read the cached data directly, otherwise read the ledger data, and finally submit the result.

For the specific flow of the cache mechanism, there is a step to check whether it is the last write operation. Explain the meaning of this operation. For the read operation, we stipulate that the data it reads must be all the data updated by the ledger after the write operation initiated earlier than it; for the write operation, different write operations for the same data can be performed asynchronously at the same time, without mutual interaction. It may be affected by this situation: two write operations W1 and W2 are initiated at the same time, of which W1 completes the update of the ledger earlier than W2. If the completion of the update of the ledger has to update the cache, this situation will occur It is possible that W2 updates the cache earlier than w1 due to network delays, which means that W1 overwrites the new results of W2, which will cause subsequent read operations to read the old results. Therefore, plus the above check operation, there are two main advantages: one is to ensure correctness, so that the read operation can read the latest results; the other is to reduce the number of updates to the cache.

6.2 Data consistency

After the introduction of the cache mechanism described above, it is possible to speed up the query of high-frequency access or data that is not modified long, preventing a large number of requests from entering the transaction process and causing delays in other operations. Then, ensuring the consistency of the data in the cache and the database is a necessary prerequisite for the cache to be effective. Among them, through the cache mechanism, although the speed of reading data can be accelerated, the above scheme does not take into account the failure of updating the cache. Imagine that if for some reason, the write transaction did not successfully update the data in the cache after the ledger was successfully updated, at this time the read request to read the cache still read the old data, and at this time the data in the database It is obviously inconsistent with the data in the cache. To this end, we add logs to manage cached updates, as shown in Figure 6. The specific scheme is as follows: (1) The write transaction successfully updates the data in the ledger; (2) The ledger database will write the operation information into the log; (3) The subscription program extracts the required data and key; (4) The subscription program attempts to delete the cache operation and finds that the deletion fails; (5) Send this information to the message queue; (6) The program obtains the data from the message queue again and retry the operation.

7 Experiments and analysis

7.1 Experiment setup

Since there are many concurrencies in the trading scenario, we implement a concurrency scenario, which can be used for trading, with a chaincode [4]. Our chaincode enables users to register their accounts, deposit, withdraw and transfer and check balances. In this paper, we mainly simulated saving money with concurrency. We use Fabric release v1.2, single peer setup running on a Lenovo T430 machine with 8GB RAM, dual core Intel i5 processor. We use a single peer but we keep the consensus mechanism turned on. We use all default configuration settings to run our experiments.

7.2 Compared methods and metrics for experiments

For the Write-Write Concurrency Conflict on Hyperledger Fabric, we compare the performance of our method LMLS with the original Hyperledger Fabric system. Although existing methods [22] [9] also work on the performance of transaction processing, their results are not comparable here, as their methods only work for transactions without concurrency conflicts. In additon, due to the original Hyperledger Fabric blockchain platform did not control concurrency for transactions with Read-Write Concurrency Conflict, this article uses a method only having read-write lock method which is containing in the cache-based method as a baseline, and compares it with our cache-based method to test the performance and effectiveness of the proposed method. We call the former RWlock_based and the latter Cache_based.

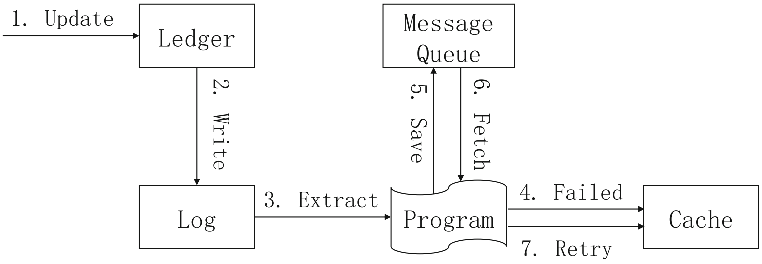

Figure 6 The log for data consistency

In this paper, we compare the performance of LMLS and Fabric with following metrics: (1) Total time - the time cost to process all transactions. (2) Success rate - the ratio of transactions successfully written to the ledger to all transactions. (3) Throughput - the amount of transactions successfully written into the ledger per unit time.

7.3 Datasets

7.3.1 Datasets for write-write concurrency conflict

For Write-Write Concurrency Conflict, we carry out experiments with three synthetically generated datasets. We implement a data generator to generate sets of transactions. In each transaction $\{username, operation, amount\}$, operation denotes the type of the transaction, such as deposit and withdrawal. The generated datasets are as follows.

- **DS1**: In this dataset, the accounts for all transactions are the same, that is, each transaction deposits for the same account. The number of transactions is 10K.
- **DS2**: In this dataset, the accounts for all transactions are not necessarily the same. The number of transactions and accounts are 10K and 1000.
- **DS3**: In this dataset, the accounts for all transactions are different, that is, each transaction deposits for different accounts. Therefore, there is no concurrency conflict in this dataset. The number of transactions is 1K.

7.3.2 Datasets for read-write concurrency conflict

Since this experiments mainly compares the problem of Read-Write Concurrency Conflict, it mainly includes read operations and write operations. Considering the complexity and uncertainty of concurrent scenarios, we randomly generate according to the proportion of control read operations and write operations. Data for experiments. Here, we define the read-write ratio R_{rw}, which is the ratio of read operations and write operations initiated in the average unit time. Assuming that the number of read operations per unit time is N_r and the number of write operations is N_w, then

$$R_{rw} = \frac{N_r}{N_w} \tag{1}$$

Each of our experiments randomly generates different transaction sets based on the read-write ratio R_{rw}. The transaction datasets we generated mainly contain two operations of querying user balance and deposit, which correspond to read operations and write operations respectively. Each transaction mainly includes the following information: (1) transaction type $(operation)$, including two types of deposit $(saveMoney)$ and query balance $(queryBalance)$; (2) transaction content, mainly including query or update Account name $(accountID)$; and the number of deposits $(amount)$.

7.4 Experimental results

7.4.1 Experiment for DS1

First, we do experiment in DS1 which the accounts for all transactions are the same. We change the transaction arrival rate, which is the average of transactions initiated per second, from 10 tps to 200 tps. Four groups of experiment are tested which with different transaction

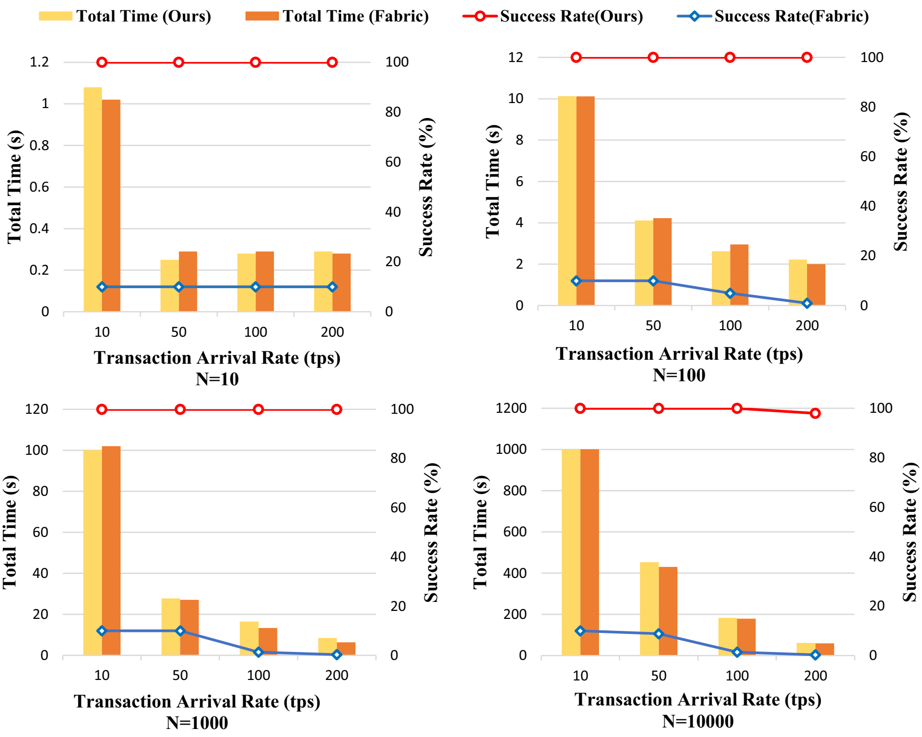

Figure 7 Time cost and success rate of LMLS and Fabric at different transaction arrival rates in DS1 (N denotes the transaction volume)

volume N of 10, 100, 1K and 10K. We test time cost and success rate, and the results can be seen from Figure 7.

As can be seen from Figure 7, on the one hand, regardless of the transactions volume, the total time of the two methods is similar, which shows that LMLS does not reduce the efficiency of the system in processing transactions. On the one hand, LMLS obviously has a higher success rate and the success rate can reach 100% no matter how high transaction arrival rate is. However, except in the case of a transaction volume of 10, with the increase of the transaction arrival rate, the success rates of Fabric have decreased significantly. Especially when the transaction arrival rate rises to 200 tps, the success rate is close to 0%. This shows that LMLS can successfully handle almost all transactions in the case of high concurrency conflicts, while Fabric cannot. Thus, LMLS is more suitable for scenarios with concurrency conflicts and the efficiency is obviously better than Fabric.

7.4.2 Experiment for DS2

To further validate the performance of our methods, we do experiment in DS2 which the accounts for all transactions are not necessarily the same. In the experiment, we initiate multiple transactions with concurrency conflicts and these transactions will modify different accounts. We define the average transaction number per user as n, the computation method

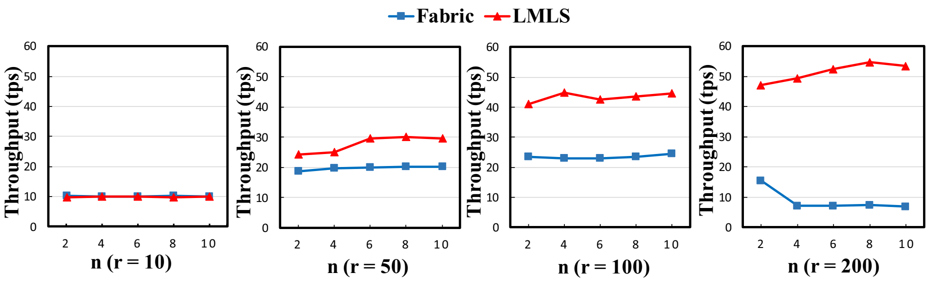

Figure 8 Throughput of LMLS and Fabric in DS2 varying n (r denotes the transaction arrival rate)

as follows:

$$n = \frac{\sum_{i=1}^{u} a_i}{u} \tag{2}$$

where u denotes the number of accounts modified in the experiment and a_i denotes the number of times the i-th account modified. For convenience, the number of times of each account modified is the same in our experiment, that is, $\forall i, j \in \{1, 2, ..., u\}, a_i = a_j$, thus, $n = a_i \ (i = 1, 2, ..., u)$. We test the throughput for two methods varying n and the results can be seen from Figure 8.

In Figure 8, n denotes the average transaction number per user and r denotes the transaction arrival rate. First, we can see that with the increase of n, the gap of throughput between Fabric and LMLS is increased. The results illustrate that LMLS is more suitable for scenarios with multiple concurrency conflicts. Second, with the increase of n, the throughput of LMLS is generally on the rise, while Fabric's unchanged, moreover, when $r = 200$, its throughput drops significantly. This shows that the efficiency of Fabric is greatly reduced in high-concurrency scenarios, but LMLS not. Third, horizontally comparing the four line charts, we can see that, as r increases, the throughput of LMLS is also increasing, which illustrates LMLS also performs well in the case of high concurrency conflicts. The experimental results show that our method is significantly more efficient than fabric in complex trading scenarios involving concurrency conflicts.

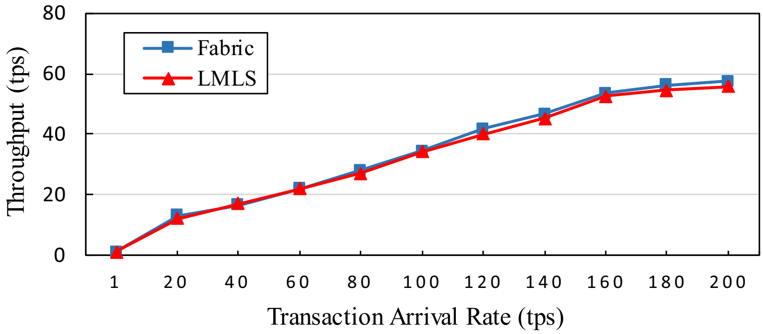

Figure 9 Throughput of LMLS and Fabric at different transaction arrival rates in DS3

7.4.3 Experiment for DS3

In order to verify the efficiency of our method in transactions without concurrent conflicts, we do experiment in DS3 which the accounts for all transactions are different, that is, no concurrency conflict in this dataset. As shown in Figure 9, we test the throughput of two methods at different arrival rates from 1 tps to 200 tps. We can see that as the transaction arrival rate increases, the throughput of Fabric as well as LMLS is increasing. In the absence of concurrency conflicts, LMLS performance is similar to Fabric. Although we add a lock mechanism, our efficiency has not decreased. The experimental results show that our methods are applicable regardless of whether there are scenarios with concurrency conflicts or without.

7.4.4 Experiments for Cache_based and RWlock_based

We performed experiments on multiple randomly generated datasets, with an average transaction volume of 10,000 transactions per dataset, and finally recorded the average throughput. Our experiment compares the RWlock_based method with the Cache_based method, and conducts experiments with the transaction arrival rate (that is, the average value of transactions initiated per second from 10tps to 200tps). Four groups of experiments were

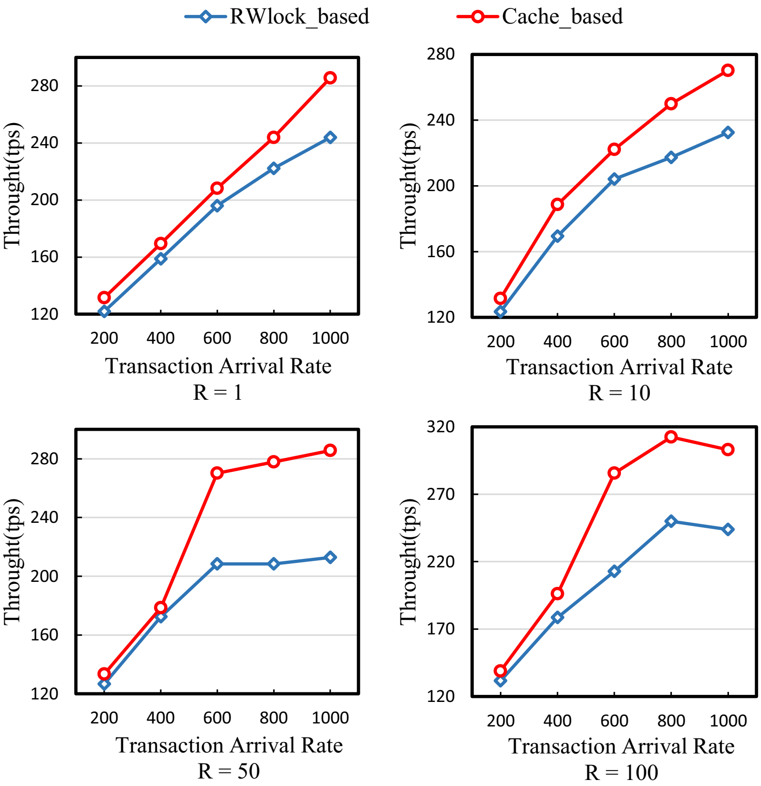

Figure 10 Throughput of RWlock_based and Cache_based with different transaction arrival rate(R denotes the read-write ratio R_{rw})

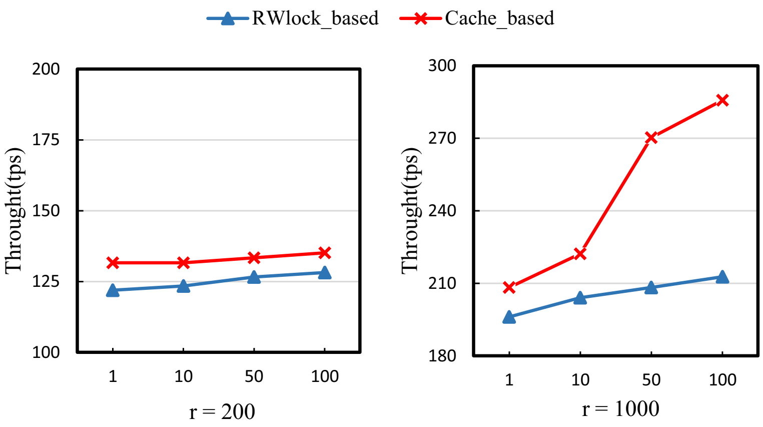

Figure 11 Throughput of RWlock_based and Cache_based with different R_{rw}(r denotes the transaction arrival rate)

performed on the read-write ratio R_{rw}, and the results can be seen in Figure 10. As can be seen from Figure 10, as the transaction arrival rate increases, the throughput of the system is greatly improved.

In addition, when the read-write ratio is 1, the throughput of the cache_ based method is only slightly higher than that of the RWlock_based method. This is because the number of read operations is low when the read-write ratio is low. It is relatively small, so the read operation waits for less time. At this time, the utilization of the cache will be reduced. Therefore, in the cache_based method, the total processing efficiency will become lower; and as the read-write ratio increases, the cache_based scheme will be significantly higher than that of RWlock_based, and the gap will be more obvious when the transaction arrival rate is higher, indicating that the cache_based method is suitable for high concurrency and read operations. When the number is greater than the number of write operations, it performs well. At the same time, the efficiency does not decrease much when the number of read operations is not large.

In addition, when the transaction arrival rates r were 200 and 1000, we perform two sets of different read-write ratios. The experimental results are shown in Figure 11. From the figure we can see that when the transaction arrival rate is low, the transaction throughput of the RWlock_based method and the cache_based method are close, and the cache_based method is slightly higher, and the throughput also increases slightly as the read-write ratio increases; when the transaction arrival rate is high, the difference in throughput between the two is more obvious, especially the larger the read-write ratio, the more obvious the gap. The experimental results also show that the method proposed in this paper has better results in scenarios with high concurrency or more frequent read operations.

7.4.5 Cost analysis

We we consider the time overhead for LMLS. On the one hand, LMLS compared to Fabric have the cost of lock-mechanism construction time. However, compared to the time it takes for the system to process the transactions, the time to build a lock is negligible, as the experimental results show. In addition, although LMLS changes the database indexing method, it does not increase the time overhead of storage. On the other hand, we analyze the storage cost of two methods. LMLS builds composite key-value pairs for each transaction with concurrency conflicts, so the number of key-value pairs on state-db increase. However, we eventually merge the composite pairs with the original pairs. Therefore, in general, storage cost has not increased. Moreover, in Hyperledger Fabric, regardless of whether the transaction is valid, it will be stored in the block if it has been sorted by Orderer. Therefore, in the case of transactions with concurrency conflicts, Fabric will package a large number of invalid transactions into blocks. In contrast, LMLS can reduce the cost of block storage.

8 Conclusion

In this paper, we focus on optimize the performance of Hyperledger Fabric by improving the handling efficiency of transactions with concurrency conflicts. We propose a novel method LMLS to optimize the performance of Hyperledger Fabric. Firstly, we design a locking mechanism to discovery conflicting transactions at the beginning of the transaction flow. Secondly, we optimize the ledger storage based on the locking mechanism, where the database indexes corresponding to conflicting transactions are changed and temporally stored in ledger. To validate the performance of the proposed solutions, extensive experiments are conducted and results demonstrate that our method outperforms the original method.

Acknowledgments This work was supported by the National Natural Science Foundation of China (Grant No. 61572335, 61572336, 61902270), and the Major Program of Natural Science Foundation, Educational Commission of Jiangsu Province, China (Grant No. 19KJA610002), and the Natural Science Foundation, Educational Commission of Jiangsu Province, China (Grant No. 19KJB520052, 19KJB520050), and Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

1. Androulaki, E., Cachin, C., Caro, A.D., Kokoris-Kogias, E.: In: López, J., Zhou, J., Soriano, M. (eds.) Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3–7, 2018, Proceedings, Part I, Lecture Notes in Computer Science, vol. 11098, pp. 111–131. Springer (2018)
2. Baliga, A., Solanki, N., Verekar, S., Pednekar, A., Kamat, P., Chatterjee, S.: In: Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20–22, 2018, pp. 65–74. IEEE (2018)
3. Bessani, A.N., Sousa, J., Alchieri, E.A.P.: In: 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23–26, 2014, pp. 355–362. IEEE Computer Society (2014)
4. Chaincodes. http://hyperledger-fabric.readthedocs.io/en/release-1.2
5. ChannelEventHub. https://fabric-sdk-node.github.io/ChannelEventHub.html
6. Dinh, T.T.A., Wang, J., Chen, G., Liu, R., Ooi, B.C., Tan, K.: In: Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., Suciu, D. (eds.) Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14–19, 2017, pp. 1085–1100. ACM (2017)
7. Ethereum blockchain app platform. https://ethereum.org/
8. Everledger: A digital global ledger. https://www.everledger.io/
9. Gorenflo, C., Lee, S., Golab, L., Keshav, S.: In: IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14–17, 2019, pp. 455–463. IEEE (2019)

10. Gupta, H., Hans, S., Aggarwal, K., Mehta, S., Chatterjee, B., Jayachandran, P.: In: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16–19, 2018, IEEE Computer Society, pp. 1489–1494 (2018)
11. Gupta, H., Hans, S., Mehta, S., Jayachandran, P.: In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 294–301. IEEE (2018)
12. Hyperledger fabric. https://www.hyperledger.org/projects/fabric
13. Nakamoto, S.: In: Bitcoin: A Peer-To-Peer Electronic Cash System (2008)
14. Nasir, Q., Qasse, I.A., Talib, M.W.A., Nassif, A.B.: Security and Communication Networks **2018**, 3976093:1 (2018)
15. Parity. https://www.parity.io/
16. Pongnumkul, S., Siripanpornchana, C., Thajchayapong, S.: In: 26th International Conference on Computer Communication and Networks, ICCCN 2017, Vancouver, BC, Canada, July 31–Aug. 3, 2017, IEEE, pp. 1–6 (2017)
17. Raman, R.K., Vaculín, R., Hind, M., Remy, S.L., Pissadaki, E.K., Bore, N.K., Daneshvar, R., Srivastava, B., Varshney, K.R.: In: IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14–17, 2019, pp. 277–284. IEEE (2019)
18. Redis. https://redis.io/
19. Securekey: Building trusted identity networks. https://securekey.com/
20. Sharma, A., Schuhknecht, F.M., Agrawal, D., Dittrich, J.: arXiv:1810.13177 (2018)
21. Sousa, J., Bessani, A., Vukolic, M.: In: 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25–28, 2018, pp. 51–58. IEEE Computer Society (2018)
22. Thakkar, P., Nathan, S., Viswanathan, B.: In: 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2018, Milwaukee, WI, USA, September 25–28, 2018, IEEE Computer Society, pp. 264–276 (2018)
23. White, M.: Digitizing global trade with maersk and ibm. https://www.ibm.com/blogs/blockchain/2018/01/digitizing-global-trade-maersk-ibm/
24. Xu, L., Chen, W., Li, Z., Xu, J., Liu, A., Zhao, L.: In: Cheng, R., Mamoulis, N., Sun, Y., Huang, X. (eds.) Web Information Systems Engineering - WISE 2019 - 20th International Conference, Hong Kong, China, November 26–30, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11881, pp. 32–47. Springer (2019)
25. Zhang, C., Xu, C., Xu, J., Tang, Y., Choi, B.: In: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8–11, 2019, pp. 842–853. IEEE (2019)

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Lu Xu[1] · Wei Chen[1] · Zhixu Li[1] · Jiajie Xu[1] · An Liu[1] · Lei Zhao[1]

Lu Xu
lxu7@stu.suda.edu.cn

Zhixu Li
zhixuli@suda.edu.cn

Jiajie Xu
xujj@suda.edu.cn

An Liu
anliu@suda.edu.cn

[1] Soochow University, Suzhou, China