# Locking Mechanism for Concurrency Conflicts on Hyperledger Fabric

Lu Xu[1], Wei Chen[1,2], Zhixu Li[1], Jiajie Xu[1], An Liu[1], and Lei Zhao[1](✉)

[1] School of Computer Science and Technology, Soochow University, Su Zhou, China
lxu7@stu.suda.edu.cn, {robertchen,zhixuli,xujj,anliu,zhaol}@suda.edu.cn
[2] Institute of Artificial Intelligence, Soochow University, Su Zhou, China

**Abstract.** 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 a novel method LMLS, which contains the following two components, 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 improve the processing efficiency. Extensive experiments conducted on three datasets 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:** Hyperledger Fabric · Concurrency · Locking 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 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 wise variety of industries. A number of blockchain platforms have been developed, including Bitcoin [16], Ethereum [3], Hyperledger Fabric [5] 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.

© Springer Nature Switzerland AG 2019
R. Cheng et al. (Eds.): WISE 2019, LNCS 11881, pp. 32–47, 2019.
https://doi.org/10.1007/978-3-030-34223-4_3

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 send 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 [18] 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 [8] and Everledger [4].

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.* [14,15] 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.* [13] 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 updating the same data in the ledger simultaneously. Since a transaction passes through multiple nodes and the transaction flow is relatively complicated, transactions with concurrency conflicts are discovered in the final step, which leads to the inefficient processing of transactions in Hyperledger Fabric.

To address above mentioned problem, we propose a novel method LMLS. 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 [7] 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. That is to say, based on LMLS, we can address concurrency conflicts in 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, we design a novel method LMLS which contains Locking Mechanism and Ledger Storage.
- 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 Sect. 2 and formulate the problem in Sect. 3. Section 4 gives a brief introduction of Hyperledger Fabric architecture. In Sect. 5 we propose LMLS method to improve the performance of Hyperledger Fabric with concurrency conflicts. In Sect. 6, experiments are conducted to validate the effectiveness of the proposed method. Finally, we conclude this paper in Sect. 7.

## 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 [12] benchmarked the popular blockchain implementations - Hyperledger Fabric, Ethereum and Parity [6] against a set of database workloads. Similar efforts include - benchmarking Hyperledger Fabric and Ethereum against transactional workloads [18]. 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.* [17] 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.* [10] 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. 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.* [14,15] present two models to optimize the temporal query performance of Hyperledger Fabric. Gorenflo *et al.* [13] 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 [11] implementation as a part of Hyperledger Fabric to improve it. Androulaki *et al.* [9] 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.* [19] 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 The Problem of Concurrency Conflicts 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. However, 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.

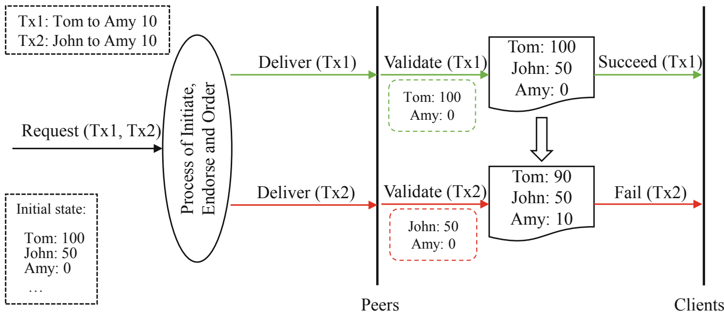


**Fig. 1.** The instance for conflicting transactions (Tx1 represents Tom transfers $10 to Amy and Tx2 represents John transfers $10 to Amy).

### 3.2 The Instance for Conflicting Transactions

Specifically, as shown in Fig. 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 some 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 shown in Fig. 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.** Given a set of transactions with concurrency conflicts in Hyperledger Fabric, a novel method LMLS is designed to tackle the problem, where a locking mechannism and the optimization of ledger storage are developed.

# 4 The Hyperledger Fabric Architecture

## 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 Fig. 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 response 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 (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,

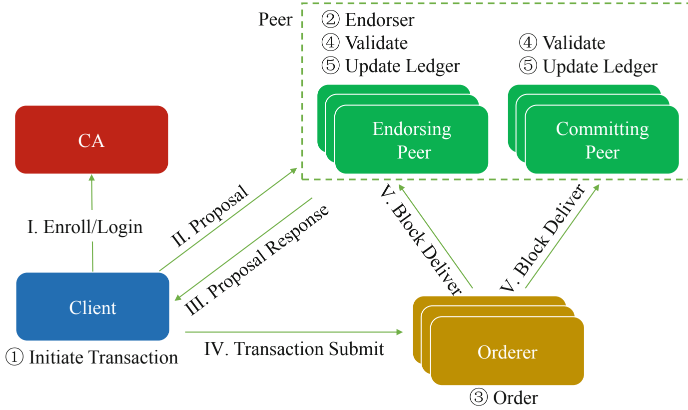


**Fig. 2.** The transaction flow of Hyperledger Fabric.

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 Updated.** 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 concurrency conflict problems 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 concurrency conflicts.

### 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 [7] 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 [2]. 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.

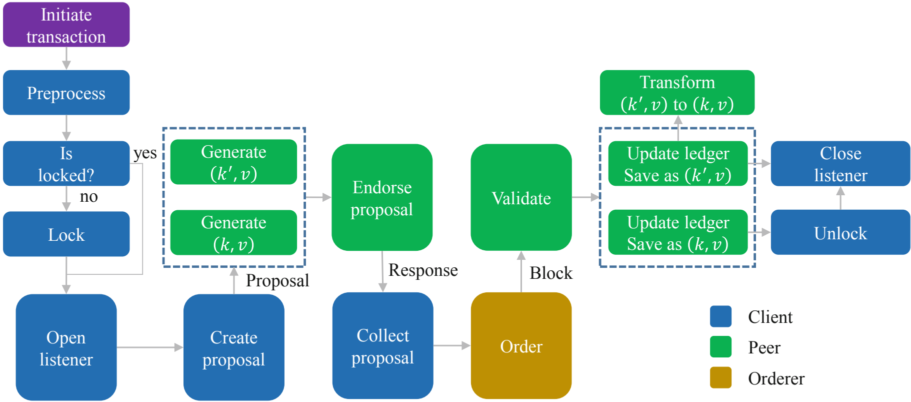


**Fig. 3.** The complete transaction flow with LMLS.

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 Fig. 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.

**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.

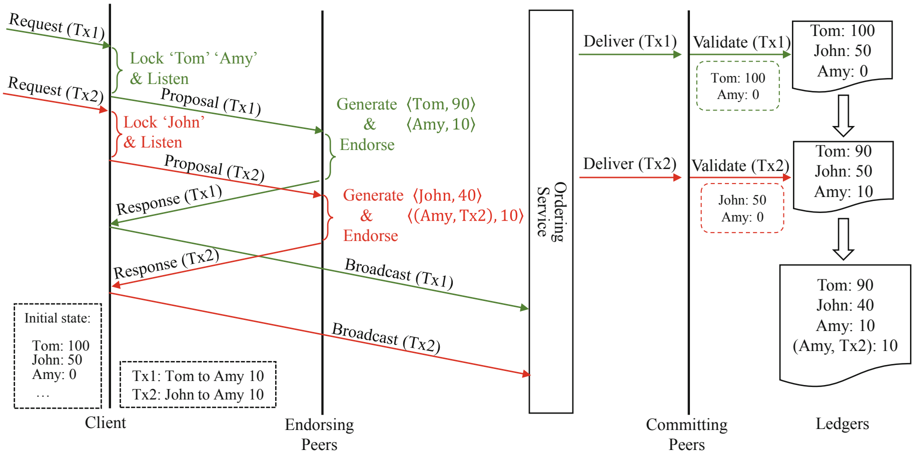


**Fig. 4.** The example for LMLS to process conflicting transactions (Tx1 represents Tom transfers $10 to Amy and Tx2 represents John transfers $10 to Amy).

### 5.4 Examples for LMLS

Continue the example in Sect. 3, we assume that Tx1 in Fig. 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 Fig. 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 Experiments and Analysis

### 6.1 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 [1]. 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 8 GB 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.

### 6.2 Compared Methods and Metrics for Experiments

We compare the performance of our method LMLS with the original Hyperledger Fabric system. Although existing methods [13,22] 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 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.

### 6.3 Datasets

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.

### 6.4 Experiment Results

**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 volume $N$ of

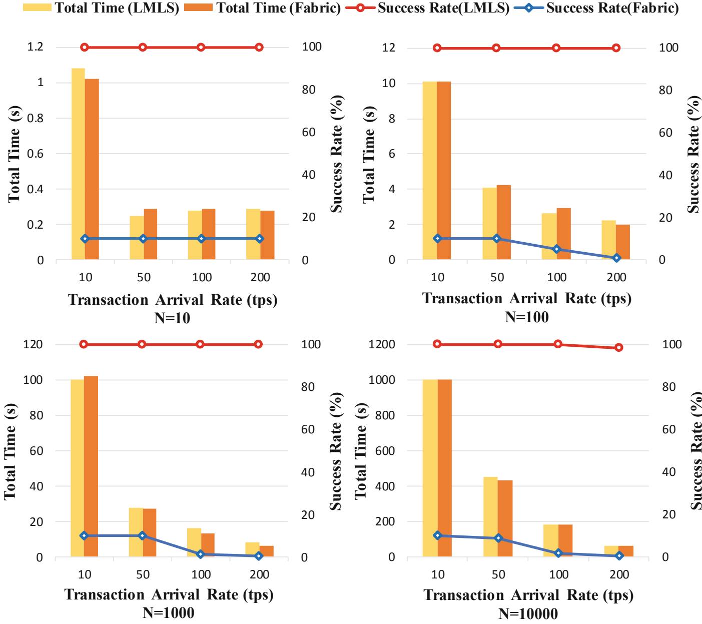


**Fig. 5.** Time cost and success rate of LMLS and Fabric at different transaction arrival rates in DS1 ($N$ denotes the transaction volume).

10, 100, 1K and 10K. We test time cost and success rate, and the results can be seen from Fig. 5.

As can be seen from Fig. 5, 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.

**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 as follows:

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

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 Fig. 6.

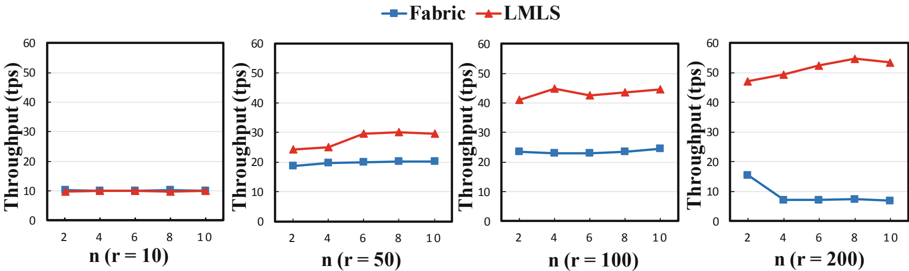


**Fig. 6.** Throughput of LMLS and Fabric in DS2 varying $n$ ($r$ denotes the transaction arrival rate).

In Fig. 6, $n$ 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.

**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 Fig. 7, 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.

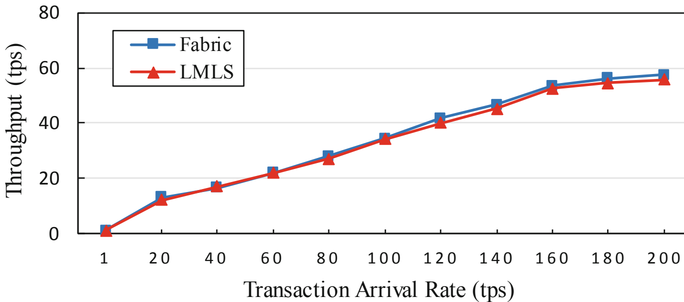


**Fig. 7.** Throughput of LMLS and Fabric at different transaction arrival rates in DS3.

**The Cost Analysis.** On the one hand, we consider the time overhead. 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.

## 7 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.

**Acknowledgements.** 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. Chaincodes. http://hyperledger-fabric.readthedocs.io/en/release-1.2/chaincode4noah.html
2. ChannelEventHub. https://fabric-sdk-node.github.io/ChannelEventHub.html
3. Ethereum blockchain app platform. https://ethereum.org/
4. Everledger: A digital global ledger. https://www.everledger.io/
5. Hyperledger fabric. https://www.hyperledger.org/projects/fabric
6. Parity. https://www.parity.io/
7. Redis. https://redis.io/
8. Securekey: Building trusted identity networks. https://securekey.com/
9. Androulaki, E., Cachin, C., De Caro, A., Kokoris-Kogias, E.: Channels: horizontal scaling and confidentiality on permissioned blockchains. In: Lopez, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018. LNCS, vol. 11098, pp. 111–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99073-6_6
10. Baliga, A., Solanki, N., Verekar, S., Pednekar, A., Kamat, P., Chatterjee, S.: Performance characterization of hyperledger fabric. In: CVCBT, pp. 65–74 (2018)
11. Bessani, A.N., Sousa, J., Alchieri, E.A.P.: State machine replication for the masses with BFT-SMART. In: DSN, pp. 355–362 (2014)

12. Dinh, T.T.A., Wang, J., Chen, G., Liu, R., Ooi, B.C., Tan, K.: BLOCKBENCH: a framework for analyzing private blockchains. In: Salihoglu, S., Zhou, W., Chirkova, R., Yang, J., Suciu, D. (eds.) SIGMOD, pp. 1085–1100 (2017)
13. Gorenflo, C., Lee, S., Golab, L., Keshav, S.: Fastfabric: scaling hyperledger fabric to 20,000 transactions per second. CoRR abs/1901.00910 (2019)
14. Gupta, H., Hans, S., Aggarwal, K., Mehta, S., Chatterjee, B., Jayachandran, P.: Efficiently processing temporal queries on hyperledger fabric. In: ICDE, pp. 1489–1494 (2018)
15. Gupta, H., Hans, S., Mehta, S., Jayachandran, P.: On building efficient temporal indexes on hyperledger fabric. In: CLOUD, pp. 294–301 (2018)
16. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
17. Nasir, Q., Qasse, I.A., Talib, M.A., Nassif, A.B.: Performance analysis of hyperledger fabric platforms. Secur. Commun. Netw. **2018**, 1–14 (2018)
18. Pongnumkul, S., Siripanpornchana, C., Thajchayapong, S.: Performance analysis of private blockchain platforms in varying workloads. In: ICCCN, pp. 1–6 (2017)
19. Raman, R.K., et al.: Trusted multi-party computation and verifiable simulations: a scalable blockchain approach. CoRR abs/1809.08438 (2018)
20. Sharma, A., Schuhknecht, F.M., Agrawal, D., Dittrich, J.: How to databasify a blockchain: the case of hyperledger fabric. CoRR abs/1810.13177 (2018)
21. Sousa, J., Bessani, A., Vukolic, M.: A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In: DSN, pp. 51–58 (2018)
22. Thakkar, P., Nathan, S., Viswanathan, B.: Performance benchmarking and optimizing hyperledger fabric blockchain platform. In: MASCOTS, 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/