IEEE ICBC'24 1570976690

Better Clients, Less Conflicts: Hyperledger Fabric Conflict Avoidance

***Abstract*—Hyperledger Fabric, a prominent permissioned blockchain platform, offers concurrent transaction processing via its Optimistic Concurrency Control (OCC) feature. However, conflicts arise when multiple transactions access the same data, severely impacting Fabric's throughput. To reduce transaction conflicts, we draw inspiration from CSMA/CA, a wireless network medium access control protocol that uses Randomized Exponential Backoff (REB) to avoid packet collisions. Based on REB, we design Fabric/CA, a client-side transaction submission protocol designed to reduce transaction conflicts. By mitigating the computational waste caused by conflicts, Fabric/CA substantially improves Fabric's performance. Our experiments show that under a high-contention workload, Fabric/CA achieves a reduction of over 98% in transaction conflicts, and boosts Fabric's goodput by more than 10x.**

***Index Terms*—Hyperledger Fabric, transaction conflicts, randomized exponential backoff**

I. Introduction

A blockchain is a distributed ledger maintained by a network of nodes, some of which may be malicious (i.e., Byzantine nodes). To ensure the seamless functioning of these decentralized systems, the majority of blockchain platforms employ Byzantine fault tolerance consensus algorithms. These algorithms *order* transactions so that all normal nodes reach consensus on the transaction order. Subsequently, these nodes *execute* transactions based on the agreed order. This transaction processing framework, which are adopted by Bitcoin [1] and Ethereum [2], is known as the *order-execute* model.

As an illustrative instance of the order-execute model, Bitcoin utilizes the Proof-of-Work (PoW) consensus algorithm for transaction ordering. After PoW determines the sequence of transactions, Bitcoin peers validate and execute transactions accordingly. For example, consider a double-spend attack where a Bitcoin client submits two transactions, T_1 and T_2, attempting to spend the same Unspent Transaction Output (UTXO). If PoW determines that T_1 precedes T_2 in the transaction order, all normal peers collectively reject T_2, thwarting the double-spend attempt successfully.

However, the order-execute model greatly limits the transaction processing speed. To ensure consistent execution outcomes across all nodes, transactions must be executed sequentially according to the agreed order. Referring back to the previous double-spend attack example, if T_1 and T_2 are executed concurrently, different nodes may have different outcomes, leading to an inconsistent state. As a result, without further adjustments, the order-execute model does not support concurrent transaction processing, limiting transaction processing efficiency.

A. Hyperledger Fabric and Transaction Conflict

To improve transaction processing efficiency, IBM introduced Hyperledger Fabric (hereafter Fabric), a permissioned blockchain solution widely embraced by enterprises [3]. Departing from the conventional order-execute model, Fabric employs a sophisticated simulate-order-validate-commit model. This distinctive approach incorporates Optimistic Concurrency Control (OCC) [4], enabling concurrent transaction processing. Fabric, like Ethereum, supports smart contracts, defining transaction logic as functions within these contracts.[1] The state of these smart contracts resides in *world state*, a versioned key-value storage system. Leveraging this versioned world state, Fabric employs Multiversion Concurrency Control (MVCC) to facilitate concurrent transaction processing.

As implied by its name, "simulate-order-validate-commit," clients first submit transactions to Fabric peers for simulation. Multiple transactions can be simulated by peers simultaneously, realizing concurrent transaction processing. It is critical to highlight that simulation results are not committed to the world state immediately. This is because different transactions may concurrently access the same key, leading to conflicting simulation results. To resolve conflicts, Fabric subsequently orders transactions, deeming a transaction valid only if its simulation result aligns with the transaction order. Valid transactions are then committed to the world state.

Specifically, to facilitate transaction validation, for each key read by a transaction, its version at the start of the simulation is stored in the transaction's simulation result. During the validation phase, transactions are deemed invalid if the version in the simulation result is lower than that in the world state, indicating that the simulation relied on outdated information. We call this type of conflicts Multiversion Concurrency Control Read Conflict (MVCCRC).[2]

MVCCRC greatly reduces Fabric's **goodput** (i.e., the number of successfully-committed transactions per second) and wastes computation and communication resources. To reduce MVCCRC, two seminal works, Fabric++ [5] and FabricSharp [6] order transactions carefully based on the transaction dependency graph. However, there are situations where transaction ordering cannot reduce MVCCRC effectively. Thus, another critical approach is to proactively abort transactions at the earliest possible stage if they risk MVCCRC. In Fabric++,

[1] In Fabric, smart contracts are commonly referred to as chaincodes. In this paper, we use these terms interchangeably.

[2] In Fabric's log messages, this type of transaction validation error is denoted as `MVCC_READ_CONFLICT`.

peers may even abort a transaction during simulation if it conflicts with another transaction that is being committed. Experimental results show that early transaction abortion avoids MVCCRC effectively and improves goodput [5].

Fabric++ and FabricSharp detect and avoid potential MVCCRC after simulation starts. However, if conflicting transactions are submitted concurrently, they still require complete simulation, wasting Fabric peers' communication and computation resources. These drawbacks are exacerbated for the following reasons:

a) After simulating a transaction, a peer has to compute a cryptographic signature on the simulation result. The computation of cryptographic signatures significantly impacts Fabric's performance [5].
b) Most transactions must be simulated by multiple peers (since peers may not trust each other). Thus, conflicting transactions often waste the resource of multiple peers.
c) Transactions that fail due to MVCCRC are often resubmitted (since the failure arises from dependencies among concurrently submitted transactions, rather than the transaction itself). Thus, MVCCRC may recur.

This paper is thus devoted to answering the following question:

Can we avoid MVCCRC even before simulation starts?

An affirmative answer to the above question mitigates the above drawbacks, improving Fabric's goodput.

Our Contribution

In this paper, we answer the above question in the affirmative, without modifications to Fabric peers or the involvement of third-party servers. Our solution draws inspiration from the Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol employed by wireless local area networks (e.g., Wi-Fi), which utilizes Randomized Exponential Backoff (REB) to avoid packet collisions in a distributed manner. In this paper, we design Fabric/CA, a client-side transaction submission control protocol that uses REB to avoid transaction conflicts. Experimental results demonstrate that under a high-contention workload, Fabric/CA reduces MVCCRC by more than 98% and increases Fabric's goodput by more than 10x.

Crucially, our approach is orthogonal to Fabric++, FabricSharp, and other Fabric optimizations such as StreamChain [7], [8] and FastFabric [9]. This inherent compatibility allows our solution to complement and seamlessly integrate with these established solutions. Moreover, unlike CSMA/CA, Fabric/CA does not require deployment on all Fabric clients. Instead, Fabric/CA only needs to be deployed on the clients of conflict-prone smart contracts. We implement `submitTransactionCA`, a wrapper function of Fabric SDK's `submitTransaction`. By calling `submitTransactionCA`, clients can apply Fabric/CA to submit transactions. Thus, the modifications required to apply Fabric/CA is minimal.

World State			Read Set		Write Set	
key	value	version	key	version	key	value
Alice	100	3	Alice	3	Alice	80
Bob	50	2	Bob	2	Bob	70

TABLE I: An example of the read set and write set for a transfer transaction from Alice to Bob of value 20.

II. Background

In Fabric, a chaincode is a collection of functions, and a transaction is a specific function execution. To initiate a transaction, one must specify the name of the function to be executed and the inputs to the function (i.e., function arguments). In addition, every transaction is governed by an endorsing policy, which specifies the legitimate peers for transaction simulation. These peers are called endorsing peers. Depending on the endorsing policy, simulation results from multiple endorsing peers might be necessary, especially when transactions involve multiple organizations.

The state of smart contracts in Fabric is stored in the world state. This world state is replicated across peers, and each peer keeps a local copy in a key-value database like LevelDB [10] or CouchDB [11]. Next, we explain the simulate-order-validate-commit model in detail.

A. The Lifetime of a Transaction

To initiate a transaction, a client submits a transaction proposal to the endorsing peers, adhering to the predefined endorsing policy. This can be done by calling the `submitTransaction` function in Fabric SDK. In this paper, we refer to the caller of `submitTransaction` as a Fabric client.

1) Simulate Transactions: Upon receiving a transaction proposal from a Fabric client, the endorsing peer evaluates the transaction against the local world state and generates a proposal response consisting of the simulation result and a cryptographic signature authenticating the simulation result. The endorsing peer then sends the proposal response to the client. Specifically, the simulation result has two sets:

a) Read Set: This set includes the keys that were read by the transaction and their corresponding versions in the local world state.
b) Write Set: This set includes all the key-value pairs that the transaction modifies or creates.

Example 1. Consider a transaction where Alice transfers a value of 20 to Bob. Assume that in peer A's local world state, Alice's balance is 100 with a version number of 3 and Bob's balance is 50 with a version number of 2. Table I then shows the corresponding read set and write set generated by peer A.

Once the client gathers all required responses in adherence to the endorsing policy, it verifies the consistency of the simulation results. If any inconsistencies arise (i.e., if different responses contain different information regarding the same key), the transaction is aborted.

2) Order Transactions: In Fabric, the critical task of transaction ordering is entrusted to a group of nodes known as orderers. Once a client successfully gathers consistent simulation results from all endorsing peers, the transaction data is sent to orderers for block creation. The transaction order within the block is determined by a consensus algorithm (e.g., Raft [12] or Kafka [13]). After a block is generated, it is broadcast to all the peers.

3) Validate and Commit Transactions: Upon receiving a block, a peer validates the transactions encapsulated within it. This validation process hinges on two fundamental mechanisms: Validation System Chaincode (VSCC) and Multiversion Concurrency Control (MVCC). VSCC checks whether the signatures associated with read/write sets obey the endorsing policy. MVCC checks whether the versions specified in the read set align with the most recent version stored in the peer's local world state. If the transaction validation fails, the transaction is marked as invalid; otherwise, the key-value pairs in the write set are committed to the peer's world state and the corresponding version numbers are increased by one.

Example 2. Consider the read/write sets in Table I again. Assume that after the read/write sets are generated, another transaction that updates Alice's balance is committed to peer A's world state (e.g., a transfer of value 30 from Chris to Alice). As a result of the commit, the version number of Alice's balance becomes 4. Thus, when the read/write sets in Table I are checked during validation, peer A detects MVCCRC as Alice's version in the read set (3) is less than that in the world state (4). Consequently, the write set in Table I cannot be committed to peer A's world state.

B. Prior Approaches on MVCCRC Reduction

To reduce MVCCRC, Fabric++ and FabricSharp apply the following two techniques from database concurrency control to Fabric:

a) **Transaction ordering:** Observe that given a set of transactions, different transaction orders yield different validation outcomes. For example, consider a block that consists of one transaction T_1 that updates a key X and 10 read-only transactions that read X. If T_1 precedes the 10 read-only transactions in the block, the 10 read-only transactions would be marked as invalid due to MVCCRC. However, if T_1 is the last transaction in the block, no MVCCRC occurs and all transactions can be committed successfully. Fabric++ and FabricSharp carefully analyze the transaction dependency graph to order transactions. During this ordering process, transactions may be aborted strategically to avoid MVCCRC.

b) **Early transaction abortion during simulation:** In Fabric++, while one transaction T_1 undergoes simulation, another transaction T_2 can be validated on the same peer concurrently. In particular, if T_1's endorsing peer detects that T_2 updates a key read by T_1, T_1 is aborted. Fabric++ achieves this through the implementation of a fine-tuned concurrency control mechanism.

C. Limitations of Prior Approaches

The effectiveness of transaction ordering diminishes when multiple transactions attempt to concurrently read and update the same key. Furthermore, in Fabric++, conflicting transactions cannot be detected and aborted during simulation if they are simulated concurrently.

Example 3. Consider a scenario with 10 transactions that concurrently read and update a key X. Regardless of the transaction order, only the first transaction is valid, and all the remaining transactions are marked as invalid due to MVCCRC. Moreover, because these transactions are simulated concurrently, conflicts cannot be detected during simulation (recall that simulation cannot alter the world state).

Because the simulation results of invalid transactions cannot be reused in the future, the communication and computation resources consumed for submitting and simulating invalid transactions are wasted. In addition, peers spend considerable effort computing cryptographic signatures on these simulation results. This resource wastage significantly impairs Fabric's efficiency.

Given the above limitations, our primary objective is thus to minimize the waste caused by conflicting transactions.[3] In particular, our aim is to prevent transaction conflicts even before transactions are submitted for simulation, thereby averting the above wasteful scenario. By doing so, the valuable communication and computation resources could be more efficiently allocated to valid transactions, consequently improving Fabric's goodput.

D. Randomized Exponential Backoff

In numerous computer science systems, the concurrent use of resources by multiple parties is restricted. For instance, in multithreaded programming, data within a critical section of code can only be accessed by one thread at a time. Similarly, in wireless networks, only one ongoing transmission is allowed, as simultaneous packet transmissions often result in collisions. In such systems, effective coordination among parties is imperative to ensure proper operation. One simple but suboptimal approach to achieve coordination is to introduce an additional server responsible for scheduling requests from multiple parties. However, this approach introduces unnecessary delays as requests must first be routed to the server before reaching the intended resource, leading to potential performance bottlenecks.

Randomized Exponential Backoff (REB) is an elegant decentralized solution that tackles the aforementioned challenge [14], [15]. In REB, when a party needs to issue a request q, it begins by generating a random number r from the interval $[0, W(q)]$, where $W(q)$ is a number associated with q. $W(q)$ is small initially (e.g., one). The party then patiently waits for r units of time before making the request. If the request fails, the party doubles the value of $W(q)$ and repeats the

[3]This objective is also one of the fundamental driving forces behind Fabric++'s design.

process. Originally designed for Ethernet networks, REB has found widespread applications in various domains of computer science, including wireless networks [16], transactional memory [17], multithread coordination [18], and congestion control [19]. In the next section, we use REB to avoid MVCCRC in Fabric.

III. Fabric/CA: Conflict Avoidance via REB

To reduce the overhead of Fabric peers, we implement REB on the client side. Originally, after a transaction request is generated by an end user (e.g., via a wallet app or an online transaction website), a Fabric client submits the transaction request to Fabric peers by calling the `submitTransaction` function in Fabric SDK. From the perspective of Fabric application developers, our solution, Fabric/CA, provides a wrapper function, `submitTransactionCA`, that incorporates REB into `submitTransaction`.

In this section, we begin by presenting a preliminary strawman design and highlighting its limitation in Section III-A. We then delve into the details of Fabric/CA in Sections III-B to III-D.

A. Strawman Design

Naturally, one might apply REB on every transaction independently. Specifically, when a transaction T is generated by an end user, the Fabric client initializes $W(T) = 1$ and proceeds as follows:

a) Generate a random number r from the interval $[0, W(T)]$.
b) Wait for r seconds.
c) Submit transaction request T to Fabric peers by calling `submitTransaction`.

If a Fabric peer throws an MVCCRC exception for T, the client then doubles the value of $W(T)$ and repeats the above three steps. This iterative process aims to adaptively adjust the waiting time for each transaction based on the extent of contention.

a) Limitation: This design is not ideal, however. While REB effectively mitigates conflicts among transactions submitted by different Fabric clients (referred to as **inter-client conflicts**), it is suboptimal in handling conflicts among transactions submitted by the same Fabric clients (referred to as **intra-client conflicts**). For instance, consider a wallet application attempting to transfer assets from the same account to multiple other accounts simultaneously. Even with the strawman design, it is possible for the client to submit these conflicting transactions concurrently, potentially leading to MVCCRC issues.

b) Overview of Fabric/CA: To effectively mitigate intra-client conflicts, a simpler and more efficient approach involves submitting conflicting transactions sequentially, bypassing the need for simultaneous REB applications. In sequential transaction submission, the client sends the next transaction only after the completion of the previous one. Consequently, transactions submitted in this manner do not conflict with each other, offering a more robust solution to the MVCCRC problem.

To realize the above idea, Fabric/CA attempts to organize transactions into different queues based on the keys they access. Transactions that access the same key are placed in the same queue. By submitting transactions from the same queue sequentially, intra-client conflicts can be avoided. To maintain a high level of parallelism, transactions from different queues can still be submitted concurrently.

B. Hot Argument and Transaction Classification

Unlike Fabric peers, Fabric clients do not install smart contracts (i.e., chaincode) and have no access to the world state. Thus, it is non-trivial to detect conflicting transactions or create the read set and write set at Fabric clients.[4] In this paper, we propose an effective approximate solution based on two observations.

Our first observation rests on the (possibly implicit) relationship between the keys accessed by a transaction and the transaction arguments[5]. To illustrate, consider the `SendPayment` function in the Smallbank chaincode [21]–[23], a benchmark utilized by Fabric++ and FabricSharp. The `SendPayment` function takes parameters `sourceAccount`, `destAccount`, and `amount`, with the intention of transferring `amount` from `sourceAccount` to `destAccount`. In this scenario, any valid `sourceAccount` value corresponds to a key-value pair in the world state, which stores the associated account balance. For example, in a `SendPayment` transaction where `sourceAccount = Alice`, the endorsing peer reads and updates Alice's balance in the world state. As a result, if two concurrent `SendPayment` calls share the same `sourceAccount` value, these two transactions cause MVCCRC. Therefore, we treat transaction arguments as keys that are accessed in the world state.

Note that different function parameters may have the same value. For example, one transaction's `sourceAccount` value may match another transaction's `destAccount` value. In such scenarios, if these two transactions are submitted concurrently, they can still cause MVCCRC. In fact, even transactions calling different functions in a chaincode can have the same argument. In the Smallbank chaincode, for example, a transaction calling `DepositChecking` may increase the balance of some account involved in another transaction calling `SendPayment`. By treating transaction arguments as the keys accessed in the world state, we can effectively detect conflicts among transactions calling different functions.

However, even if two transactions share the same argument, they may not cause conflicts. This is because some function parameter cannot be blamed for transaction conflicts. For example, even if two `SendPayment` calls have the same value of `amount`, as long as they access different accounts, they

[4]While Fabric clients receive the read set and write set from Fabric peers *after* transaction simulation, the simulation is wasteful if the transaction causes conflict. To avoid such waste, we aim to detect conflicts *before* transaction simulation. Moreover, in some optimized Fabric, transactions will be sent directly to the ordering service, even before simulation is completed [9], [20]. In this case, the client cannot abort conflicting transactions.

[5]Following the convention in most programming languages, we refer to the values of transaction parameters as transaction arguments.

cannot cause MVCCRC. This shows that to detect conflicts, we should not consider all function arguments.

To address this issue, we leverage the second observation: the presence of *hot accounts* in many transfer scenarios. Informally, this refers to the situations where the majority of transactions involve only a few specific accounts, following the 80/20 rule. For instance, consider a scenario where numerous users attempt to purchase tickets from a ticket vendor. All the resulting `SendPayment` transactions share the same `destAccount` value (i.e., the vendor). This forms a many-to-one scenario, where various sources transfer funds to a common recipient. In such cases, the value of `destAccount` can be viewed as a hot account. Another prevalent scenario is the one-to-many case where a single source transfers funds to multiple distinct recipients. In such cases, the value of `sourceAccount` can be viewed as a hot account.

Based on the above two observations (i.e., the presence of hot accounts and treating arguments as keys), we introduce the concept of *hot argument*. In Fabric/CA, every transaction has a hot argument, denoted as `hotarg`. Transactions sharing the same `hotarg` will be grouped into the same queue for sequential submission. For instance, in a scenario where a Fabric client receives numerous `SendPayment` transactions sharing the same `destAccount` value (i.e., a many-to-one transfer scenario), an appropriate `hotarg` would be the value of `destAccount`. Similarly, in a one-to-many scenario, an appropriate `hotarg` would be the value of `sourceAccount`.

`hotarg` can be left empty if a transaction is unlikely to cause conflicts. For example, an auditing application may periodically submits transactions to check account balances during off-peak hours.[6] These transactions can set `hotarg = NULL`. If a transaction has no `hotarg` (i.e., `hotarg = NULL`), it can be submitted immediately.

C. Selection of `hotarg`

The selection of `hotarg` is trivial for transactions that involve only one account. For example, in `DepositChecking`, where a given amount is added to a given account, `hotarg` is the value of the account. Thus, we mainly focus on transactions that involve multiple accounts. Recall that we have categorized conflicts into intra-client conflicts and inter-client conflicts, and the strawman design, which applies REB to each transaction independently, effectively mitigates inter-client conflicts. Observe that most intra-client conflicts are caused by one-to-many transfers. According to the previous discussion, an appropriate `hotarg` in a one-to-many scenario is the value of `sourceAccount`. Thus, in this paper, we set `hotarg` as the value of `sourceAccount`. In general, to mitigate intra-client conflicts, among all transaction arguments, `hotarg` should be the one associated with the transaction generator.

[6]One can opt not to persist read-only transactions on the ledger by calling `evaluate` in Fabric SDK. Evaluated transactions cannot cause conflicts. However, some applications (e.g., audit-related processes) require read-only transactions to be recorded on the ledger, making `evaluate` inappropriate.

Remark 1. Alternatively, one may determine `hotarg` in a dynamic manner. For instance, when end users generate transactions at a high rate, these transactions could be batched at the Fabric client. `hotarg` can then be determined on a per-batch basis by analyzing the transaction dependency graph. Even advanced techniques such as machine learning could be utilized to predict the appropriate `hotarg` for each transaction. Our experimental results demonstrate that using our static setting of `hotarg`, Fabric/CA already outperforms the vanilla Fabric. As such, the refinement of `hotarg` selection is left for future exploration.

D. Interface of `submitTransactionCA`

For the `submitTransaction` function in Fabric SDK, the input parameters consist of `name` (representing the transaction function name) and `args` (an array indicating the transaction function arguments). Our customized wrapper function, `submitTransactionCA`, extends this interface by adding the new parameter, `hotarg`.

Note that in most cases, Fabric clients do not need to parse `args` to extract `hotarg`, and end users do not need to specify `hotarg`. To obtain `args`, Fabric clients usually provide end users with a user-friendly interface (e.g., an HTML form or an iOS/Android application), displaying the necessary parameters for the transaction and the corresponding fields where end users can enter the arguments. Fabric clients can then extract the desired `hotarg` from the appropriate field or from the information of the end user.

E. Implementation of `submitTransactionCA`

To implement `submitTransactionCA`, we create a transaction queue for each `hotarg`. After a transaction request is made by calling `submitTransactionCA`, the transaction, which consists of the function name and arguments, is stored in the queue corresponding to its `hotarg`. If the transaction has no `hotarg` (i.e., `hotarg = NULL`), we directly submit the transaction to Fabric peers by invoking `submitTransaction`. For each transaction queue, we submit the first transaction in the queue by invoking `submitTransaction`.

We stress that we do not remove the first transaction in a transaction queue (i.e., dequeue) until the client receives a successful commit response or the number of resubmissions exceeds a predefined threshold. Thus, transactions with the same `hotarg` cannot be processed by Fabric peers concurrently, mitigating intra-client conflicts. Transactions with different `hotarg` are stored in different transaction queues and thus can still be processed by Fabric peers concurrently.

If a transaction is aborted due to MVCCRC, we resubmit it with REB (Algorithm 1). Observe that when all transactions have no `hotarg`, Fabric/CA degenerates into the aforementioned strawman design. To ensure a good user experience and avoid excessive delays in transaction resubmission, the maximum value of $W(T)$ is set to 16 in out implementation.

During our experimentation, we observe that Fabric peers may crash due to transaction processing overloading. We

Algorithm 1: Transaction Resubmission With REB

input: An aborted transaction T

1 $resubmissionCount[T]$++

2 $W(T) \leftarrow \min\left(2^{resubmissionCount[T]}, 16\right)$

3 $r \leftarrow$ a random number from the interval $[0, W(T)]$

4 Wait for r seconds

5 Submit T to peers by calling `submitTransaction`

thus enhance our implementation with congestion control on the client side. Our congestion control is similar to that in TCP [24], and we omit the details due to the space constraint. We define outstanding transactions as transactions that have been submitted to Fabric peers by `submitTransaction`, but have not been successfully committed. Thus, for every transaction queue, it has at most one outstanding transaction. To invoke `submitTransaction`, a Fabric client has to wait until the number of outstanding transactions is less than `cwnd`. We update `cwnd` using the slow start and congestion avoidance mechanisms in TCP. This combined strategy aims to ensure both the efficiency and stability of the Fabric network.

IV. Evaluation

Our experiment is built on Fabric 2.2, with Smallbank as the chosen chaincode. We deploy Fabric peers through Docker on two physical machines, each of which has an Intel i7-12700H CPU and 32 GB RAM. The default settings are shown in Table II. We run four isolated clients on an Apple M2 Pro. Each client constantly generates 500 transactions per second (excluding the resubmissions of failed transactions). These physical machines are connected by a wired LAN.

The functions in the Smallbank chaincode and their `hotarg` are shown in Table III. We use Zipfian distributions to select the accounts involved in these functions. Note that when the Zipfian skewness is zero, it degenerates into a uniform distribution. In the following discussion, we refer to the Zipfian skewness as the account skewness. Transaction functions are selected based on the workload type, which can be categorized as read-heavy, write-heavy, or balanced:

a) In a **read-heavy** workload, the majority (95%) of transactions are read-only (i.e., Query), with only a small percentage (5%) involving write operations (i.e., the remaining five functions in Table III).

b) In a **balanced** workload, the distribution of transactions is evenly split between read-only transactions (50%) and transactions that update the world state (50%).

c) In a **write-heavy** workload, the majority (95%) of transactions update the world state, with only a small percentage (5%) being read-only.

We compare Fabric/CA with the aforementioned strawman design (hereafter referred to as **Fabric/REB**) and ordinary Fabric clients, which resubmit a failed transaction immediately without REB (referred to as **Fabric** in Fig. 1 and Fig. 2). To have a fair comparison, we employ the congestion control introduced in Section III-E on all three client designs. In particular, if a client wants to invoke `submitTransaction` (e.g., to resubmit a failed transaction), it has to wait until the number of outstanding transactions is less than `cwnd`. At a high level, Fabric/REB is built on ordinary Fabric clients by adding REB (Fabric/REB = Fabric + REB), and Fabric/CA is further built on Fabric/REB by using `hotarg` to classify transactions (Fabric/CA = Fabric + REB + `hotarg`).

Parameter	Value
Fabric version	2.2
Chaincode	Smallbank
Endorsement policy	Majority
Number of organizations	2
Number of peers per organization	3
Database	LevelDB
Block size	100
Number of accounts	10,000
Number of clients	4
Transaction generation rate	500 txs per second per client
Zipfian skewness	0-2 (0.2 increment)
Experiment duration	60 seconds

TABLE II: Fabric Settings

Function	`hotarg`
`Query`	value of `account`
`DepositChecking`	value of `account`
`WriteCheck`	value of `account`
`TransactSavings`	value of `account`
`SendPayment`	value of `sourceAccount`
`Amalgamate`	value of `sourceAccount`

TABLE III: Functions in the Smallbank chaincode and their `hotarg`. For the first four functions, only one account is involved in the function, and thus that account is `hotarg`.

Fig. 1 (left) shows that, compared to ordinary clients and Fabric/REB, Fabric/CA has significantly less MVCCRC. The effectiveness of Fabric/CA lies in its dual approach: it reduces inter-client MVCCRC through REB and reduces intra-client conflicts by sequentially submitting transactions with the same `hotarg`. Specifically, when the account skewness is over 1.4, Fabric/CA reduces MVCCRC by 98% under the read-heavy workload, compared to ordinary clients. Observe that Fabric/REB cannot reduce MVCCRC in high-contention scenarios, because a client may send conflicting transactions concurrently. In contrast, Fabric/CA employs `hotarg` and transaction queues to address this issue. Importantly, the reduction in MVCCRC allows Fabric peers to allocate more resources to less conflict-prone transactions, resulting in a more efficient resource allocation.

For Fabric/CA, the number of MVCCRC is maximized when the account skewness is around one. This is because when the account skewness is smaller, there are less conflicting transactions. On the other hand, when the account skewness is higher, there are less distinct `hotarg` values, which reduces the number of transaction queues. Given that the transaction submission rate is limited by the number transaction queues, Fabric/CA has less MVCCRC when the account skewness is high, such as the case where the account skewness is two.

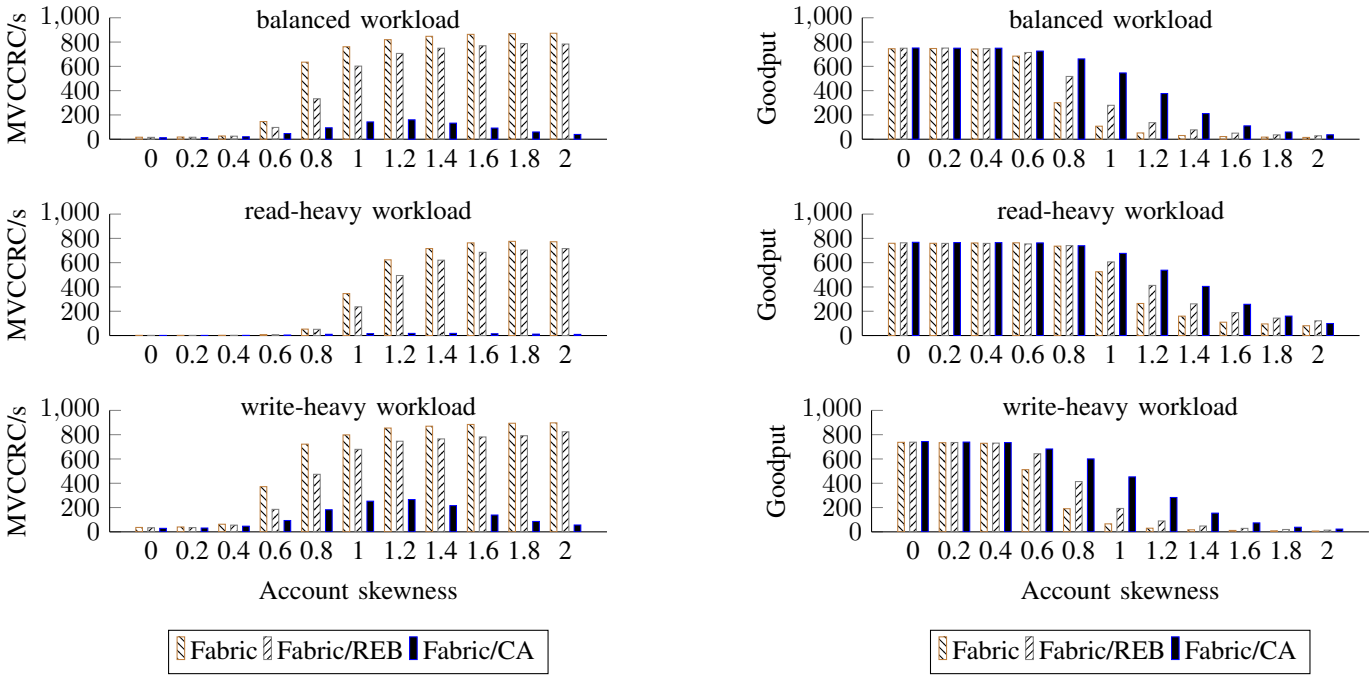

Fig. 1: Impact of different client-side solutions on MVCCRC (left) and goodput (right) under various workloads.

Account skewness	0.0	0.4	0.8	1.2	1.6	2.0
Latency (second)	0.44	0.46	1.23	3.10	4.05	3.75

TABLE IV: Transaction latency under the balanced workload.

Table IV provides additional insights into the average transaction latency of Fabric/CA under the balanced transaction workload. The latency of a transaction is defined as the duration between the invocation of `submitTransaction` (or the detection of MVCCRC on the client side) and the reception of the commit response. Thus, both REB's random waiting time and Fabric peers' processing times contribute to the latency. The results show that, thanks to REB, Fabric/CA can dynamically adjust its waiting time based on the extent of contention. It is important to note that transaction latency can be further reduced by decreasing $W(T)$.

The reduction in MVCCRC translates into improved goodput. At an account skewness of one under the balanced workload, Fig. 1 (right) shows that Fabric/REB improves the goodput of ordinary clients by 2.61x, and integrating `hotarg` further improves Fabric/REB's goodput by 1.95x. This result demonstrates the importance of both REB and `hotarg`. By combining this two techniques, Fabric/CA improves the goodput of ordinary clients by at least 3x in the write-heavy workload when the account skewness is over 0.8. Notably, at an account skewness of 1.2, the improvement exceeds 9x.

Fig. 1 shows that when the account skewness is over 1.6, the goodput is low. This is because under such settings, almost all transactions are conflicting, and thus the goodput is inevitably low. However, these settings may be too pessimistic and thus we next simulate a scenario where a subset of transactions is less likely to conflict with others. Specifically, we select the `account` value in `Query` functions uniformly randomly, but we still select accounts involved in other functions (e.g., `sourceAccount` in `SendPayment`) by Zipfian distribution. As a result, `Query` is less likely to cause conflicts, and we set `Query`'s `hotarg` as `NULL`. Note that functions other than `Query` are still conflict-prone when the account skewness is high. Moreover, we simulate a geographically distributed network by adding a network delay of 100 ms between nodes.

The results are shown in Fig. 2. Compared to Fig. 1, Fabric/CA demonstrates a substantial improvement in goodput. Because ordinary clients and Fabric/REB still submit many conflicting transactions, their subsequent resubmissions and the newly generated conflicting transactions soon saturate the whole system.[7] As a result, more resources are allocated to conflicting transactions, leading to a poor goodput. Fabric/CA, on the contrary, submits much less conflicting transactions and reduces MVCCRC significantly. As a result, Fabric peers can allocate more resources to transactions that are less likely to cause conflicts (e.g., `Query`). Notably, with an account skewness of at least 1.6, the improvement exceeds 10x.

V. Related Work

Prior works have proposed various strategies to address the issue of transaction conflicts. These strategies can be divided into two groups: peer-side solutions and client-side solutions.

[7]Even if we grant ordinary clients an unfair advantage of not resubmitting failed transactions (labelled as Fabric w/o resubmissions in Fig. 2), Fabric/CA still improves the goodput by 1.3x when the account skewness is over 1.6.

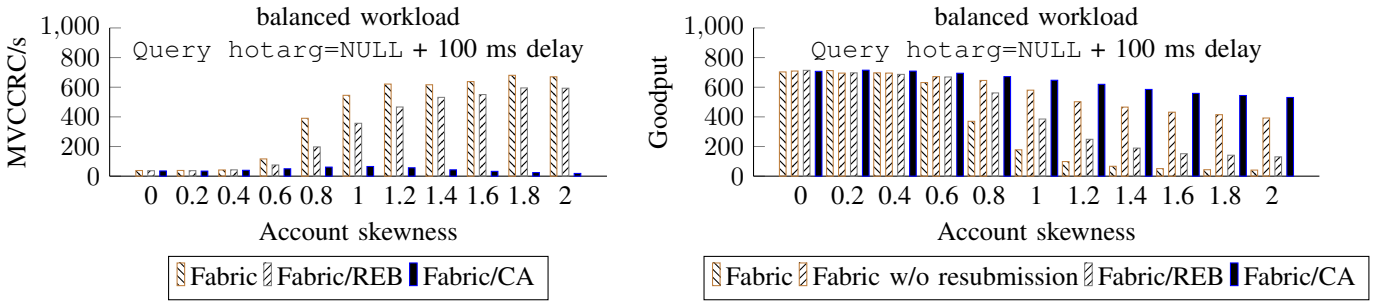

Fig. 2: Impact of different client-side solutions on MVCCRC (left) and goodput (right) under the balanced workload with addition network delay and `Query hotarg=NULL`.

A. Peer-Side Solutions

Transaction Reodering: Sharma et al. [5], Ruan et al. [6], and Sun and Yuan [25] proposed transaction reorder algorithms that analyze the dependencies among transactions, either on a per-transaction or per-block basis. However, when conflicting transactions read and write the same key, transaction reordering cannot eliminate transaction conflicts effectively.

Additional Lock mechanism: Xu et al. [26] and Xu et al. [27] adopted Redis-based lock mechanisms to prevent transaction conflicts. During transaction execution, locks are applied to the keys being accessed to prevent other transactions from using them, thereby avoiding transaction conflicts. However, these additional distributed lock mechanisms introduce significant communication overhead and may unnecessarily increase transaction latency.

Transaction Latency Reduction: István et al. [7], [8] proposed StreamChain, an optimized Fabric that processes transactions in a stream fashion rather than in batches (i.e., blocks). StreamChain significantly reduces transaction latency in data center networks with the help of an FPGA-based ordering service. Chacko et al. [28] demonstrated that due to the substantial decrease in transaction latency, StreamChain effectively avoids MVCCRC. However, StreamChain targets at datacenter networks where Fabric peers are connected by high-bandwidth low-latency links. Thus, it is not best suitable for geographically distributed networks. A similar work is FastFabric [9], which mainly improves the efficiency of transaction ordering and validation. Our work is orthogonal to StreamChain and FastFabric.

Transaction Re-execution: Gorenflo et al. [29] proposed XOX Fabric, which re-executes conflicting transactions after conflicts are detected in the validation phase. However, to efficiently re-execute a transaction, a patch-up code must be added to the chain code.

B. Client-Side Designs

The above peer-side designs significantly alter the implementation of the official Hyperledger Fabric. Enterprises may prefer the official versions provided by Hyperledger Foundation, as they offer better stability, security, and support [30]. Thus, client-side methods may be a more appealing option.

Chacko et al. [31] implemented transaction rate control on the client side. Clients reduce the transaction submission rate when the proportion of invalid transactions is high within a certain time period. However, the proposed rate control is applied universally on all transactions, regardless of their dependencies. This may unnecessarily decrease the throughput since some transactions may not conflict with others. Zhang et al. [32] aimed to reduce intra-client conflicts by examining the simulation results on the client side. Transactions that cause intra-client conflicts will not be sent to the ordering service. Instead, they will be sent back to Fabric peers for new simulation. However, in some optimized Fabric, transactions may be sent to the ordering service even before simulation is completed (e.g., [9], [20]). In such cases, the proposed method in [32] cannot avoid conflicts. Moreover, the design in [32] did not resolve inter-client conflicts on the client side.

VI. Concluding Remarks

Most existing solutions to reduce Fabric transaction conflicts require substantial modifications to Fabric peers. However, such modifications may not be applicable due to stability, security, support, and compatibility issues. While direct modifications to Fabric peers are constrained, application developers have the flexibility to shape their own client interfaces through websites or mobile applications. This paper demonstrates that a proper client design can effectively mitigate transaction conflicts without modifying Fabric peers.

It may be tempting to think that Fabric/CA is effective only when it is adopted by *all* Fabric clients. However, since transactions of different chaincodes cannot conflict with each other, Fabric/CA can be implemented in a per-chaincode basis. Specifically, only the clients associated with a conflict-prone chaincode need to adopt Fabric/CA.

This paper does not consider transactions priorities. When transaction conflicts arise, prioritizing urgent transactions becomes crucial. Recent server-side solutions have successfully achieved transaction prioritization under optimistic concurrency control [33], [34]. We believe that transaction priority can also be realized on the client side using REB (e.g., different priorities have different ways of generating the random waiting time), and this is an important future work.

REFERENCES

[1] "Bitcoin: A peer-to-peer electronic cash system," https://bitcoin.org/bitcoin.pdf.

[2] "Ethereum whitepaper," https://ethereum.org/en/whitepaper/, 2023.

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[4] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[5] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: the case of hyperledger fabric," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 105–122.

[6] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 543–557.

[7] Z. István, A. Sorniotti, and M. Vukolić, "Streamchain: Do blockchains need blocks?" in *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, ser. SERIAL'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–6. [Online]. Available: https://doi.org/10.1145/3284764.3284765

[8] L. Kuhring, Z. István, A. Sorniotti, and M. Vukolić, "Streamchain: Building a low-latency permissioned blockchain for enterprise use-cases," in *2021 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2021, pp. 130–139.

[9] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2019, pp. 455–463.

[10] "Leveldb," https://github.com/google/leveldb.

[11] "Couchdb," https://couchdb.apache.org/.

[12] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[13] "Apache kafka," https://kafka.apache.org/.

[14] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM*, vol. 19, no. 7, pp. 395–404, 1976.

[15] M. A. Bender, J. T. Fineman, S. Gilbert, and M. Young, "Scaling exponential backoff: constant throughput, polylogarithmic channel-access attempts, and robustness," *Journal of the ACM (JACM)*, vol. 66, no. 1, pp. 1–33, 2018.

[16] J. F. Kurose, *Computer networking: A top-down approach featuring the internet, 3/E*. Pearson Education India, 2005.

[17] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on Computer architecture*, 1993, pp. 289–300.

[18] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE, 2001, pp. 294–305.

[19] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM computer communication review*, vol. 18, no. 4, pp. 314–329, 1988.

[20] J. Qi, X. Chen, Y. Jiang, J. Jiang, T. Shen, S. Zhao, S. Wang, G. Zhang, L. Chen, M. H. Au, and H. Cui, "Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 18–34. [Online]. Available: https://doi.org/10.1145/3477132.3483574

[21] https://github.com/ooibc88/FabricSharp/blob/master/benchmark/smallbank/smallbank.go.

[22] "Smallbank transaction family," https://sawtooth.hyperledger.org/docs/1.2/transaction_family_specifications/smallbank_transaction_family.html.

[23] https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/.

[24] V. Paxson, M. Allman, and W. Stevens, "Tcp congestion control," 1999.

[25] Q. Sun and Y. Yuan, "Gbcl: Reduce concurrency conflicts in hyperledger fabric," in *2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS)*. IEEE, 2022, pp. 15–19.

[26] L. Xu, W. Chen, Z. Li, J. Xu, A. Liu, and L. Zhao, "Solutions for concurrency conflict problem on hyperledger fabric," *World Wide Web*, vol. 24, pp. 463–482, 2021.

[27] Z. Xu, D. Liao, X. Dong, H. Han, Z. Yan, and K. Ye, "Lmqf: Hyperledger fabric concurrent transaction conflict solution based on distributed lock and message queue," in *2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, 2023, pp. 1855–1860.

[28] J. A. Chacko, R. Mayer, and H.-A. Jacobsen, "Why do my blockchain transactions fail? a study of hyperledger fabric," in *Proceedings of the 2021 international conference on management of data*, 2021, pp. 221–234.

[29] C. Gorenflo, L. Golab, and S. Keshav, "Xox fabric: A hybrid approach to blockchain transaction execution," in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2020, pp. 1–9.

[30] V. Capocasale, D. Gotta, and G. Perboli, "Comparative analysis of permissioned blockchain frameworks for industrial applications," *Blockchain: Research and Applications*, vol. 4, no. 1, p. 100113, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2096720922000549

[31] J. A. Chacko, R. Mayer, and H.-A. Jacobsen, "How to optimize my blockchain? a multi-level recommendation approach," *Proc. ACM Manag. Data*, vol. 1, no. 1, may 2023. [Online]. Available: https://doi.org/10.1145/3588704

[32] S. Zhang, E. Zhou, B. Pi, J. Sun, K. Yamashita, and Y. Nomura, "A solution for the risk of non-deterministic transactions in hyperledger fabric," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2019, pp. 253–261.

[33] C. Ye, W.-C. Hwang, K. Chen, and X. Yu, "Polaris: Enabling transaction priority in optimistic concurrency control," *Proc. ACM Manag. Data*, vol. 1, no. 1, may 2023. [Online]. Available: https://doi.org/10.1145/3588724

[34] L. Yang, X. Yan, and B. Wong, "Natto: Providing distributed transaction prioritization for high-contention workloads," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 715–729. [Online]. Available: https://doi.org/10.1145/3514221.3526161