TxChain: Scaling Sharded Decentralized Ledger via Chained Transaction Sequences

Zheng Xu[1,2], Rui Jiang[1,2], Peng Zhang[1,2], Tun Lu[1,2](✉), and Ning Gu[1,2]

[1] School of Computer Science, Fudan University, Shanghai, China
{zxu17,jiangr20,zhangpeng_,lutun,ninggu}@fudan.edu.cn
[2] Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China

Abstract. The blockchain has become the most prevalent distributed ledger (DL). Sharding has emerged as a major solution to the scalability bottleneck of DLs. From the underlying data structure of existing sharding schemes, although the Directed Acyclic Graph (DAG)-based topology improves the scalability of DLs compared to chained blocks, the security and reliability of consensus mechanisms in DAG-based DLs have not been verified. Moreover, these schemes suffer from high communication overhead when scaling out. To address these issues, we propose a sharded DL named TxChain, which adopts a novel data structure manipulated by the unit of transaction and constituted by chained transaction sequences of each account. TxChain optimistically processes concurrent transactions and ensures the consistency of all shards via transaction sequence conversion (TSC)-based consensus mechanism. Shards maintain the full replica of TxChain and execute transactions by trustworthy validators, which reduce the frequency of communication with other shards. We theoretically prove the consistency of shards maintained by TSC and demonstrate TxChain's throughput scales with low latency through extensive experiments.

1 Introduction

Blockchain makes the distributed ledgers (DLs) evolve into a irreversible and decentralized data maintenance technology. Each node in the DL serially processes generated transactions, however, the performance of a single node has become the throughput bottleneck of the DL. Sharding has become a feasible and proven horizontal scaling approach which assigns nodes to multiple shards to handle a portion of transactions and enables parallelization of the computation and storage in DLs for high throughput. Maintainers of shards process transactions in their local shard and update the state of the entire system.

The underlying data of existing sharded DLs is in the structure of chained blocks or the Directed Acyclic Graph (DAG). Although DAG-based data structure can improve the scalability of DLs, it is difficult and unstable to achieve the final consistency. Moreover, mainstream sharded DLs such as RapidChain [8],

A. Bhattacharya et al. (Eds.): DASFAA 2022, LNCS 13245, pp. 333–340, 2022.
https://doi.org/10.1007/978-3-031-00123-9_27

OmniLedger [4] and Elastico [5] use the unspent transaction outputs (UTXO)-based transaction model which has weak programmability, high computational complexity and large storage redundancy. The Byzantine fault-tolerant (BFT) consensus mechanism in these DLs may cause high communication costs when the system scales up. These above problems indicate that existing sharded DLs still have a lot of room for improvement in terms of the data structure and parallel processing performance.

Based on the above problems and challenges, we propose a novel DL named TxChain. To achieve high scalability, TxChain processes each transaction unit parallelly by selected validators based on sharding technology. Validators execute intra-shard transactions instantly in the local shard and broadcast them to remote shards. All shards store a full replica of TxChain and collaboratively maintain their states by synchronizing transactions in accounts' TxSEQs. To ensure the consistency of all shards, three kinds of transaction dependencies and a transaction sequence conversion (TSC)-based consensus mechanism are proposed to determine the ordinal relationship between transactions. Based on this consensus mechanism, TxSEQs can have the same transaction order among different shards in accordance with the original dependency. Moreover, to optimize the latency of processing transactions, honest and trustworthy validators are selected according to their stake ratio and are protected by the trusted execution environment (TEE), which reduce the communication overhead of TxChain compared to BFT-based DLs. Besides, the modification of each shard can be completely and consistently recorded in each replica, thus a transaction can be confirmed in a low time delay without querying the state of other shards repeatedly. We summarize our main contributions as follows:

- We propose a novel sharded DL named TxChain which is parallelly manipulated by the unit of transaction and constituted by accounts' transaction sequences.
- We propose a consensus mechanism based on the transaction sequence conversion (TSC) to maintain transaction dependencies and consistency among shards. TxChain's high scalability with low confirmation latency has been demonstrated via extensive experiments.

2 System Overview and Problem Definition

2.1 System Model

Nodes are assigned to different shards $\mathcal{S} = \{S_1, S_2, \cdots, S_n\}$. Validators are selected according to their stake ratio in the local shard (the validator in shard S_1 is represented by V_1). Nodes with higher stakes are more likely to be selected. The selected validator verifies transactions, consents on the order of them, broadcasts them to remote shards and maintains the consistency of TxChain on behalf of nodes in the shard. All validator are protected by the TEE, and have no intention to be evil because of the stake-based selection. Then combining with the settings and assumptions in prior sharding-based blockchains [1,8], all validators can be regard as honest and trustworthy. TxChain adopts the structure of Merkle Patricia tree (MPT), but its leaf node is one account composed of chained transaction sequences. Figure 1 shows the data structure of TxChain. Intra-shard transactions are executed instantly in the local replica, and then are broadcast to remote shards to synchronize the account status. When transactions are received by a remote shard, the validator first verifies the signature of each transaction to ensure its validity, and then orders and executes each transaction.

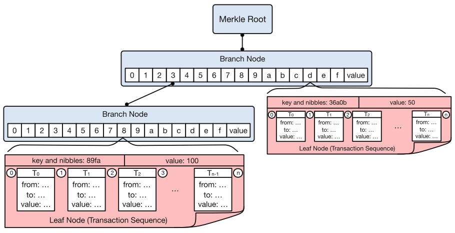

Merkle Root
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
key and nibbles: 36a0b
value: 50
T0 from: ... to: ... value: ...
T1
T2
Tn
Leaf Node (Transaction Sequence)
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
key and nibbles: 89fa
value: 100
T0 from: ... to: ... value: ...
T1
T2
Tn-1
Leaf Node (Transaction Sequence)

Fig. 1. Data structure of TxChain.

2.2 Transaction Model

TxChain adopts a state-based account whose state is cumulative results of related transactions. The transaction is defined as $\langle from, to, value, s, p, ts, fee \rangle_\sigma$ where $from$ is the sender of the transaction T, to is the recipient of T, $value$ is the transaction amount, s is the number of the shard generating T, p is the position of T in the TxSEQ of the generating account, ts is the timestamp, fee is the transaction fee, and σ is the signature of T.

In addition to executing local transactions, the validator V also needs to execute the received remote transaction T by inserting them into the TxSEQ of the specified account. Before executing T, the validator V needs to determine its relative position in the TxSEQ. V has to determine the ordinal relationship between transactions, which is determined by their dependencies. It is worth noting that only transactions in the same account constitute a dependency. All shards cannot have the exact same time in the decentralized environment, which means that it is impossible to directly determine the orders of transactions from different shards. Therefore, in order to determine the position of T in the TxSEQ and maintain consistency of its order in all shards, this paper proposes transaction dependencies. T_a and T_b are from shards S_i and S_j respectively and are executed on the same account, and their dependencies are defined as follows:

Definition 1. ***Dependencies.*** *T_a and T_b satisfy the causality relationship "$\succ$" iff (1) $T_a.ts < T_b.ts$ when $S_i = S_j$, or (2) T_a has executed in S_j before T_b generates when $S_i \neq S_j$; T_a and T_b satisfy the concurrency relationship "$\|$" iff neither $T_a \succ T_b$ nor $T_b \succ T_a$; T_a and T_b satisfy the conflict relationship "$\otimes$" iff $T_a \parallel T_b$ and $T_a.p = T_b.p$.*

2.3 Problem Definition

Transactions are executed instantly in the local shard and then are broadcast to remote shards for synchronization. After the remote shard receives the transaction T, the shard's state may change and it cannot execute T directly in the current view according to the position specified by $T.p$. As shown in Fig. 2, V_1 attempts to append T to T_2 in the TxSEQ of account "0x3889fa", i.e., $T.p = 3$. V_1 then sends T after S_2. However, V_2 has concurrently executed the transaction $T_2^{'}$ at the position 2 of account "0x3889fa" because $T_1 \succ T_2^{'} \succ T_2$, which changes $T_2.p$ from 2 to 3. Thus, T should be ordered at position 4 which is after $T_2.p$ in S_2. If T is executed directly in S_2 based on $T.p = 3$, the transaction dependencies are violated and TxSEQs maintained among shards will be inconsistent.

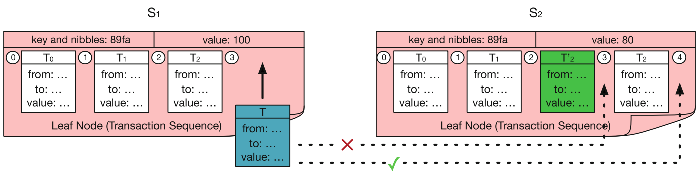


Fig. 2. Inconsistency example in TxChain.

TxChain supports the parallel execution of transactions in all shards. However, shards' state may change after transactions reach remote shards, resulting in violations of transaction dependencies and inconsistent shard states. Therefore, how to design a consensus mechanism which executes transactions in obedience to their dependencies and maintains the consistency of all shards becomes the key problem faced by TxChain.

3 Consensus Mechanism in TxChain

3.1 Prerequisites of Transaction Execution

The consensus in TxChain is to order and execute transactions based on transaction dependencies and ensures the consistency of these orders among shards. In order to maintain the transaction dependencies, TxChain adopts the vector-based timestamp [2] to determine the execution order of transactions. Each shard in TxChain has a collection of accounts $\mathcal{L} = \{L_1, L_2, \cdots, L_m\}, L_j \in \mathcal{L}$. In addition to the local state of the accounts, each shard needs to record the state of other shards. Each shard's state is composed of all the account states in it. We use the execution vector $EV_{S_i}^{L_j}$ to denote the state of L_j in S_i. Each element of EV represents the state of the same account in different shards and starts with 0, i.e., $EV_{S_i}^{L_j}[S_i] = 0, S_i \in \mathcal{S}$. After the shard S_j executes a transaction from S_i, L_j's execution vector $EV_{S_j}^{L_j}[S_i] = EV_{S_j}^{L_j}[S_i]+1$. The timestamp of a transaction

T is $T.ts$ which represents the EV when it is generated. When remote shards receive T, they maintain the dependencies and execution order of transactions according to $T.ts$. Therefore, based on the timestamp of transactions, we present how to determine whether a transaction can be instantly executed. Meanwhile, The consistency constraints of TxChain should make transactions match their causality relationship and let transactions with higher transaction fee be ordered in the front.

Definition 2. ***Transaction Execution.*** *A transaction T can be executed instantly in the shard iff (1) T is from this local shard, or (2) T is from the remote shard and transactions which is causally before T have been executed.*

3.2 Transaction Sequence Conversion Algorithm

To solve the problem in Sect. 2.3, according to the consistency constraints of TxChain, we design the consensus mechanism based on the transaction sequence conversion (TSC) algorithm. T_a and T_b are two concurrent transactions for the same account L in S_i and S_j respectively, and are executed instantly in their local shards. When T_b arrives at S_i, the execution vector of L in S_i is $EV^L_{S_i}$. The ordering position of T_b in S_i shifts because the execution of T_a, so T_b cannot be executed directly at the $T_b.p$ in S_i. Therefore, TSC refers to the idea of address space transformation (AST) [3,7] to convert the account's TxSEQ to a specific state, and then execute transactions according to Definition 2.

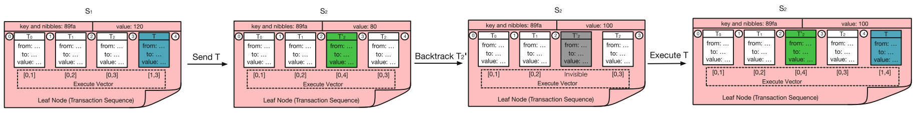


Fig. 3. Transaction sequence conversion.

We still use the example in Sect. 2.3 for illustrating the process of TSC, which has been presented in Fig. 3. Before V_1 attempts to execute T in the account $L =$ "$0x3889fa$", the execution vector of L is $EV^L_{S_1} = [0, 3]$. When T arrives at S_2, $EV^L_{S_2} = [0, 4]$. TSC traces S_2 back to the state when T is generated, i.e., $EV^L_{S_2} = T.ts = [0, 3]$. In other words, TSC sets the impact of $T_2^{'}$ to be invisible. After T is executed in S_2, the $EV^L_{S_2}$ is updated to $[1, 4]$. Finally, TSC restores the impact of $T_2^{'}$. Similarly, S_1 follows the above steps to synchronize $T_2^{'}$.

Based on the above content, TSC is presented as Algorithm 1. The transaction T is generated from account L of the remote shard S_j and it will be executed in L of S_i whose execution vector is $EV^L_{S_i}$. The first step of TSC is to verify the signature of T for its integrity. Next, TSC traces the TxSEQ of L back to the state when T is generated, i.e., adjusts $EV^L_{S_i}$ to equal $T.ts$. The process of backtrack determines which transactions have been executed when $EV^L_{S_i} = T.ts$. For one transaction T' in the TxSEQ of L maintained by S_i, if

Algorithm 1. The Transaction Sequence Conversion Algorithm

```
Input: Transaction sequence $TxSEQ$, transaction $T$;
1: if Verify ($T$) then
2:     Backtrack ($TxSEQ, T.ts$)
3:     Execute $T$ in $TxSEQ$
4:     $EV_{S_i}^{L}[S_j] = EV_{S_i}^{L}[S_j] + 1$
5:     Backtrack $\left(TxSEQ, EV_{S_i}^{L}\right)$
6: else
7:     Abort $T$
8: end if
```

Algorithm 2. The Sequencing Algorithm

```
Input: Sequencing interval $(p_x, p_y)$, transaction $T$
Output: Ordering position $Site$
1: $Site \leftarrow null$
2: for $T'.p$ between $(p_x, p_y)$ do
3:     if $T \parallel T'$ then
4:         if $TOrder(T) < TOrder(T')$ && $Site = null$ then
5:             $Site \leftarrow T'.p$
6:         end if
7:     else if $T \succ T'$ then
8:         $Site \leftarrow T'.p$
9:         break
10:    end if
11:    $T' \leftarrow T'.next$
12: end for
13: if $Site = null$ then
14:     return $T'.p$
15: else
16:     return $Site$
17: end if
```

$T.ts \geq T'.ts$, the impact of T' is set as visible, otherwise the impact of T' is set as invisible. The comparison of timestamp complies with the following rules. If $T'.ts = T.ts$, each corresponding element in $T'.ts$ and $T.ts$ is exactly the same. If $T'.ts < T.ts$, each corresponding element in $T'.ts$ is not greater than $T.ts$, and the sum of the elements in $T'.ts$ is less than $T.ts$. If $T'.ts > T.ts$, $T'.ts$ contains at least one corresponding element greater than $T.ts$.

After TSC has traced the shard to the state when T is executed, the ordering position of T in the TxSEQ has been determined, which is the interval between two positions. However, due to the backtrack process, there may be several invisible transactions in that interval. Therefore, it is also necessary to compare the relationship between T and these invisible transactions to finally confirm the ordering position of T. In order to compare the relationship between T and these invisible transactions, we can also use $TOrder$ [6] in addition to transaction dependencies defined in Sect. 2.2.

Definition 3. ***TOrder.*** *Given two transactions T_a and T_b, $TOrder(T_a) < TOrder(T_b)$ iff (1) $SUM(T_a.ts) < SUM(T_b.ts)$, or (2) $T_a.fee > T_b.fee$ when $SUM(T_a.ts) = SUM(T_b.ts)$, or (3) $i < j$, when $SUM(T_a.ts) = SUM(T_b.ts)$ and $T_a.fee = T_b.fee$. Meanwhile, $SUM(T.ts) = \sum_{i=1}^{n} T.ts[S_i]$.*

We have known T will be inserted between two positions p_x and p_y. Then we traverse each transaction T' in the sequencing interval (p_x, p_y). If T' and T are concurrent and $TOrder\,(T) < TOrder\,(T')$, $T'.p$ becomes the ordering position of T. If T causally precedes T', $T'.p$ is the ordering position of T and the traversal process ends. The process of ordering transactions in (p_x, p_y) can be described as Algorithm 2. Then T will be executed according to its ordering position.

4 Performance Evaluation

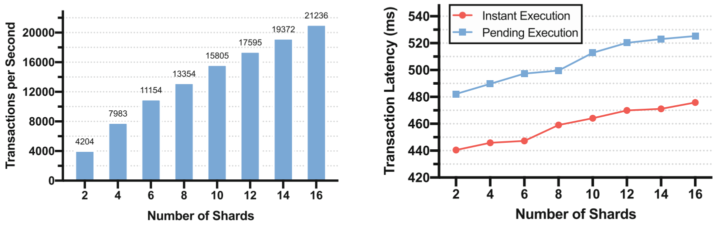
4.1 Experimental Setup

The prototype of TxChain is implemented in Golang, and its nodes spread across 16 machines. Every machine is equipped with the Intel Xeon E5128 CPU, 32 GB of RAM, 10 Gbps network link and Ubuntu 16.04 LTS operating system. In order to emulate the realistic network environment, we limit the bandwidth between nodes to 20Mbps and artificially insert a delay of 100 ms to communication links.

4.2 Throughput Scalability and Transaction Latency of TxChain

We evaluate the throughput scalability and transaction latency of TxChain in terms of the number of shards in TxChain $|\mathcal{S}|$.

As shown in Fig. 4(a), the throughput of TxChain is improved when $|\mathcal{S}|$ increases. When we double $|\mathcal{S}|$, the throughput of TxChain can be increased by 1.59 to 1.89 times. Compared to RapidChain [8] which achieves 1.57 to 1.70 times when doubling its network size, TxChain can achieve higher scalability. TxChain have better scalability especially when $|\mathcal{S}|$ is small because each shard maintains the full replica of TxChain and TSC has greater computational overhead when the amount data in TxChain expands.



(a) Throughput Scalability (b) Transaction Latency

Fig. 4. Performance of TxChain.

Figure 4(b) plots the latency of processing a transaction which can be executed instantly or not respectively for different $|\mathcal{S}|$. Transactions which cannot be executed instantly is pending for a ordering position by TSC. We take the execution of a pending transaction as the example and find that the latency increases slightly from 482.26 milliseconds (ms) to 525.31 ms when $|\mathcal{S}|$ varies from 2 to 16. With the same $|\mathcal{S}|$, a pending transaction has roughly 10% more latency than the transaction can be executed instantly. These results show that the time delay is mainly due to TxSEQs conversion in TSC.

5 Conclusion

In this paper, we propose TxChain which is a novel DL based on sharding and constituted by accounts' transaction sequences. TxChain enable shards to update the global state via the unit of transaction in parallel, which greatly scales its throughput. The transaction sequence conversion (TSC)-based consensus mechanism traces accounts' transaction sequences to the state when transactions can be ordered correctly for maintaining the shards' consistency. We implement TxChain and demonstrate its high scalability and low transaction latency.

Acknowledgements. This work was supported by the Science and Technology Commission of Shanghai Municipality under Grant No. 21511101501 and the National Natural Science Foundation of China (NSFC) under Grant No. 61932007.

References

1. Dang, H., Dinh, T.T.A., Loghin, D., Chang, E.C., Lin, Q., Ooi, B.C.: Towards scaling blockchain systems via sharding. In: Proceedings of the 2019 International Conference on Management of Data, pp. 123–140 (2019)
2. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, pp. 399–407 (1989)
3. Gu, N., Yang, J., Zhang, Q.: Consistency maintenance based on the mark & retrace technique in groupware systems. In: Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work, pp. 264–273 (2005)
4. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: Omniledger: a secure, scale-out, decentralized ledger via sharding. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 583–598. IEEE (2018)
5. Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., Saxena, P.: A secure sharding protocol for open blockchains. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 17–30 (2016)
6. Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Trans. Comput.-Hum. Inter. (TOCHI) **5**(1), 63–108 (1998)
7. Yang, J., Wang, H., Gu, N., Liu, Y., Wang, C., Zhang, Q.: Lock-free consistency control for web 2.0 applications. In: Proceedings of the 17th international conference on World Wide Web, pp. 725–734 (2008)
8. Zamani, M., Movahedi, M., Raykova, M.: Rapidchain: scaling blockchain via full sharding. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 931–948 (2018)