DecoupleChain: A Two-Layer Blockchain Sharding System Enabling Frequent Shard Reconfiguration

Huawei Huang, Miaoyong Xu, Chenlin Wu, Xiaofei Luo, Jian Zheng, Jianru Lin, Zibin Zheng
SSE, Sun Yat-Sen University, China. Corresponding author: Zibin Zheng.
GuangDong Engineering Technology Research Center of Blockchain.

***Abstract*—Sharding is a promising technique for scaling out blockchains. For example, Ethereum introduced dank-sharding in its upgrade EIP-4844. However, how to guarantee the security of each network shard remains a challenge. In this paper, we present DecoupleChain, a two-layer blockchain sharding system, which implements the *state sharding* in a Layer2 blockchain and adopts a Layer1 blockchain as a trusted reference for Layer2's state ledger. By decoupling the functions of Layer2 shards into consensus and storage, and developing a reliable inter-layer data verification method based on Merkle proofs, DecoupleChain can perform low-overhead frequent shard reconfiguration during system running. Experiments using real-world transactions demonstrate that the median reconfiguration latency of our method is 35% lower than *tMPT* and 67% lower than *Fast sync*. When reconfiguration occurs every 12 seconds, Ethereum's TPS decreases to 75% or even lower, while DecoupleChain's TPS remains stable without significant fluctuations.**

***Index Terms*—Blockchain, Sharding, Reconfiguration**

I. Introduction

Blockchain trilemma [1] suggests that the security, decentralization, and scalability of a blockchain cannot be achieved simultaneously. Classical blockchains like Bitcoin [2] and Ethereum [3] prioritized security and decentralization but sacrificed scalability. To meet the demands of high transaction throughput and low transaction latency, Luu et al. [4] proposed the first blockchain sharding protocol. The core idea of sharding is to partition the blockchain network into multiple segments, each of which is called a shard. By allowing multiple shards to verify and execute transactions in parallel, a sharded blockchain can achieve higher throughput and scalability. Subsequent studies on sharding, including OmniLedger [5], Rapidchain [6], and Monoxide [7], have extended the technique of blockchain sharding. Nowadays, advanced sharding approaches widely adopt *state sharding* [8], in which accounts and transactions are allocated to different network shards. The advantage of state sharding is that it reduces the storage overhead for each shard because all shards can collaboratively amortize a proportion of the entire state of all accounts.

Even though sharding technology offers promising prospects for large-scale blockchain applications, it currently faces challenges. Public blockchains typically operate in unregulated environments. When profitable, attackers might employ various tactics such as scams, DDoS, and bribery attacks [9], [10], etc., to conquer network nodes within a blockchain. Attackers could interfere and even manipulate the consensus process. We denote the nodes controlled by attackers as malicious nodes.

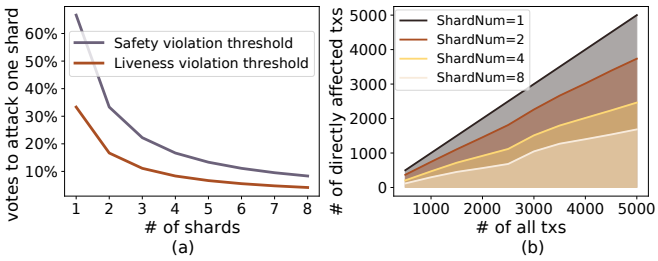


Fig. 1. Motivation experiments. Subfigure (a) illustrates the malicious voting power that can violate the *liveness* property or even the *safety* property of the consensus in a single shard. Subfigure (b) shows that the number of affected transactions increases when a single shard is continuously under attack.

Motivation. In PBFT consensus [11], on the one hand, if the voting weight of malicious nodes exceeds 1/3, malicious nodes can violate the *liveness* property. On the other hand, when their voting weight surpasses 2/3, the *safety* property of the consensus will be violated. In sharded blockchains, since a single shard only consists of a small proportion of the entire blockchain nodes, it becomes easier to attack blockchain shards from the standpoint of malicious nodes [7], [12].

We use Ethereum historical data to simulate how malicious nodes threaten the safety and liveness properties of a blockchain sharding system. Assuming that all blockchain shards exploit PBFT protocol as their local consensus, Fig. 1(a) illustrates that as the number of shards increases, the voting power required by malicious nodes to violate the consensus of a shard declines. This observation implies that the malicious nodes' attack threshold drops following the growing number of blockchain shards. When other consensus protocols are adopted in blockchain shards, similar trends can be observed [7]. Fig. 1(b) shows that as a shard continually is being under attack, the number of transactions directly affected within the entire system keeps increasing. Due to the existence of cross-shard transactions, when malicious nodes can violate the *safety* of a specific shard and arbitrarily modify its ledger state, the states in other shards will be impacted, too. These malicious behaviors bring a significant threat to the overall security of the sharding blockchain.

To ensure the security of sharded blockchains, previous studies [7], [8] have adopted *shard reconfiguration*. The idea behind shard reconfiguration is to periodically shuffle the

consensus nodes in all shards randomly, thereby preventing malicious nodes from concentrating in a particular shard. However, since those nodes scheduled to migrate to other shards need to duplicate the local state from their new shards during the shard reconfiguration, shuffling nodes across shards would introduce significant overhead of state duplication. As a result, the frequency of shard reconfiguration is set typically not high, often in days, according to the literature [6]. This long period of shard reconfiguration provides opportunities for malicious nodes. Thus, this paper raises the following research question (RQ).

RQ: *Is it possible to significantly reduce the overhead of shard reconfiguration, such that the sharded blockchain system can always maintain a high level of security by performing frequent shard reconfiguration?*

To answer this question, this paper presents a two-layer blockchain sharding system named *DecoupleChain*, which consists of two blockchain layers, namely Layer1 and Layer2. In Layer2 of DecoupleChain, we implement a new state sharding by decoupling the storage and consensus functionalities. Specifically, we divide all blockchain nodes into two categories of shards, i.e., *storage shards* and *consensus shards*. The consensus shards are *stateless*, which means that they do not store state ledger data. When a consensus shard needs to verify a transaction, the shard will retrieve the target account's state and its associated validity proof from the corresponding pairwise storage shard. When a bunch of transactions complete execution in the consensus shard, the shard sends a new block to the storage shard, which then stores the state of the related accounts associated with these transactions.

The Layer1 of DecoupleChain employs a trusted third-party public chain (e.g., Ethereum). The consensus shards of Layer2 store the metadata of each block in a smart contract deployed in the Layer1 chain. When a node in Layer2 needs to verify the validity of an account state, a transaction, or a block, it can retrieve the relevant metadata from this smart contract. The Layer1 blockchain can also offer randomness sources for some synchronization actions (such as shard reconfiguration) in Layer2.

Using the proposed two-layer architecture, consensus shards can perform low-overhead frequent reconfiguration, which can improve the robustness and security of the blockchain sharding system. Our study makes the following **contributions**.

- **Originality.** We propose a two-layer blockchain system, named *DecoupleChain*, which enables low-overhead frequent shard reconfiguration to enhance the security of blockchain sharding.
- **Methodology.** DecoupleChain decouples the storage and consensus by delegating them into different shards. We then proposed a correctness-verification method for validating accounts' state and transaction receipts, as well as a timeout mechanism aiming to ensure the atomicity of transaction execution.
- **Implementation.** We have implemented a prototype of DecoupleChain and conducted experiments using real-world transactions on multiple physical machines. The results indicate that even under frequent reconfiguration, DecoupleChain's throughput is hardly affected. In contrast, Ethereum's TPS decreases to 75% or even lower.

II. Related Work

A. Security of Blockchain Sharding

Although the security of shards can be enhanced through reconfiguration methods [4], Conflux [12] reports that reconfiguration can be only conducted at extended intervals, due to the substantial overhead. This provides attackers with ample time to concentrate their efforts on a specific shard. Rapidchain [6] introduces the principle of limited cuckoo reconfiguration, which reallocates new nodes and retains some old nodes. This can effectively prevent the attacker's leave-rejoin attack. However, its defense against bribery attacks is limited and can only handle slowly-adaptive adversaries [13], [14]. Moreover, the reconfiguration process is controlled by the reference committee, posing a risk of centralization. tMPT [15] takes the advantage of the presence of hot accounts in blockchain transactions. During reconfiguration, it only synchronizes the states of hot accounts, thus reducing the synchronization overhead for nodes. However, during the transaction execution by nodes, the missing account states must rely on witness shards for execution provision. Furthermore, these witness shards also control the reconfiguration process, leading to a risk of centralization. CoChain [16] incorporates the discovery and the handling of malicious shards on top of regular shard reconfiguration. In its system, each shard is supervised by a so-called CoC Group comprised of multiple shards. The CoC conducts cross-shard consensus on the intra-shard consensus results of a shard. If this shard is found to be malicious, it is replaced. However, when malicious nodes manipulate over 2/3 of a shard's nodes, CoC will fail to detect this situation. Additionally, this cross-shard consensus involves more complex communication and synchronization mechanisms, making the system hard to scale.

Distinct from reconfiguration-based solutions, Monoxide [7] introduces Chu-ko-nu mining, in which each miner is no longer confined to a specific shard but can participate in block production across multiple shards, thereby ensuring shard security. However, this design increases the resource requirements for miners, making it unfavorable for low-resource nodes to participate. Thus, this manner degrades the degree of the blockchain's decentralization [17].

In summary, these approaches either have not properly addressed the security issues of sharded blockchains, or they have sacrificed some scalability or decentralization in the process of security enhancement.

B. Decoupling Blockchain Functions

Modular blockchain [18] proposes the idea of decoupling various functions of a blockchain. After decoupling, each layer implements a part of the blockchain's functionality, such as data availability, transaction execution, consensus, etc. The advantage of modularization is that different modules can perform their specific duties, providing higher flexibility,

scalability, and maintainability. There are also related works on decoupling shard functions in sharded blockchains. For example, Jenga [19] selects some nodes from multiple shards to construct an execution channel, instructing them to execute transactions that invoke smart contracts. This design eliminates the cross-shard communications during state acquisition, thereby improving system efficiency. W3Chain [20] partitions the correctness of transaction data into block correctness and chain consistency.

In contrast, we propose a design that decouples shard consensus and shard storage. In our design, the stateless consensus shards can be frequently reorganized to guarantee consensus security.

III. Preliminaries

A. Layer1 and Layer2

A Layer2 blockchain refers to a blockchain that works through collaboration with a mainstream blockchain such as Ethereum [3] or Solana [21]. These major mainstream blockchains are often referred to as *Layer1 chains* or *main chains*. Layer2 blockchains offer faster transaction processing and lower transaction fees than Layer1 blockchains. Thus, Layer2 solutions can reduce the burden in the Layer1 blockchain in terms of transaction handling. During the execution of a Layer2 blockchain, a summary of Layer2 transactions will be submitted to Layer1 periodically for a permanent record.

In contrast to the goal of improving throughput [22], we introduce a two-layer architecture in DecoupleChain primarily for data verification. Since storage shards are not frequently reconfigured in our design, the data they provide may not be fully trusted. To address this issue, a trusted blockchain is implemented in Layer 1, which is responsible for storing and retrieving block metadata. Any node in Layer 2 can verify the integrity and validity of the data with the support of Layer 1.

B. Merkle Patricia Trie

Merkle Patricia Trie (MPT) is a data structure proposed by Ethereum and widely adopted by many blockchains nowadays. It can store key-value pairs, providing the holy grail of $O(\log(n))$ efficiency for inserts, lookups, and deletes. Fig. 2 is an example of using MPT to store accounts. MPT contains three types of nodes: extension nodes, branch nodes, and leaf nodes. Each node stores the hash value of its child nodes.

In the context of blockchains, MPT is commonly used to serve the following data structures: *state tree*, *transaction tree*, and *receipt tree*.

Due to the unique properties of MPT, when there is a need to verify whether the balance of a particular account is correct, validators do not need to know the complete state tree. Instead, they only need a brief *Merkle Proof* from a storage node that possesses the complete state tree. The Merkle Proof corresponding to an account consists of the root node of the state tree, the leaf node where the account resides, and the other nodes along the proof path between the root and leaf nodes. For instance, the Merkle Proof for account `c573135` is

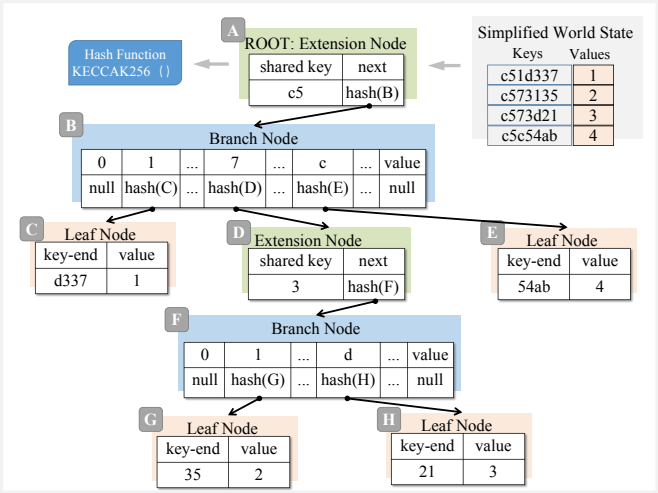


Fig. 2. An example of using MPT to store account states in a blockchain. Letters `A`, `B`, `C`, `D`, $\cdots$, `H` labeled in the upper left corner of each square serve as identifiers for the nodes of MPT.

[`A, B, D, F, G`], while for account `c5c54ab`, the Merkle Proof is [`A, B, E`]. Upon receiving the Merkle Proof, the validator starts hashing from the leaf node upward through each node on the proof path and compares each hash with those in the Merkle Proof. If any hash does not match, the verification fails. If the hash of the root node matches the one in the Merkle Proof, the verification is successful.

MPT can be used to prove both the existence and non-existence of a given account. For example, [`A, B, C`] can prove that an account with the key `c51d336` does not exist; otherwise, `C` would not be a leaf node. We refer to the Merkle Proof that demonstrates the existence of a certain account (or a transaction) as *proof of inclusion*. Similarly, the Merkle Proof that demonstrates the non-existence of a certain account (or a transaction) is referred to as *proof of exclusion*.

Given the non-reversibility of hashes, and provided that the root of the state tree is known, a Merkle Proof cannot be made up. As a result, any validator can quickly verify the state of an account with only a minimal amount of data. In the same manner, a Merkle Proof can be used to verify whether a transaction has been packed into a block or whether it has been executed successfully.

DecoupleChain employs MPT for its state tree, transaction tree, and receipt tree.

IV. System Design of DecoupleChain

A. System Overview

The architecture of the proposed DecoupleChain system is shown in Fig. 3. DecoupleChain consists of two layers, each running its own blockchain.

The blockchain in Layer2 is a dedicated sharded blockchain consisting of multiple shards, with each shard running the PBFT consensus locally. Different from the traditional state sharding, we decouple the two functions of traditional shards, i.e., *storage* and *consensus*, and assign them to two types of

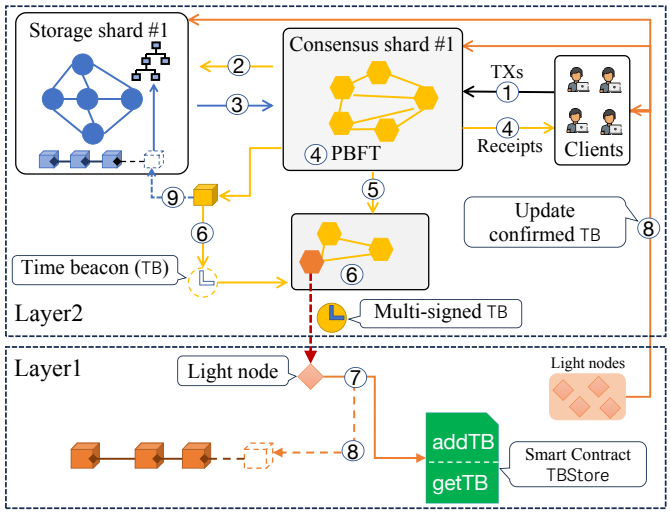


Fig. 3. An overview of the proposed two-layer *DecoupleChain* system. The operation steps are depicted as follows. Step ①: Clients submit/broadcast transactions. Step ②: A consensus shard requests account states from a storage shard. Step ③: A storage shard returns states and Merkle proofs. Step ④: The consensus shard runs PBFT consensus to generate a new block and returns receipts to clients. Step ⑤: The consensus shard elects a subset of nodes through VRF [23]. Step ⑥: Time beacon (abbr. as TB) is extracted from the block and multi-signed by the elected nodes, then relays to Layer1. Step ⑦ Light node in Layer1 proposes a transaction to record that TB on the smart contract. Step ⑧: Light nodes wait until the TB is confirmed, then update TB to the nodes of Layer2 that subscribe to it. Step ⑨: The storage shard commits the block on the blockchain and finalizes ledger states to the state tree.

shards, respectively. Storage shards are responsible for storing account states, blocks, and other data. Consensus shards handle transaction verification, packaging, and consensus. Each consensus shard is bound to a storage shard. This means that the consensus shard processes transactions related to the accounts in the associated storage shard. For easy understanding, the mutually bound consensus and storage shards share the same ID.

The blockchain in Layer1 can be any reliable third-party blockchain, such as Ethereum [3]. A dedicated smart contract (named `TBStore`) is deployed in Layer1, and nodes within the Layer2 blockchain interact with Layer1 through this contract `TBStore`. To directly obtain the latest state of the contract or initiate transactions to invoke the contract, operator of each Layer2 node can choose to set up a light node in the Layer1 blockchain (this method is recommended). Alternatively, this task can be delegated to a trusted third-party node, which forwards messages. In addition, the system also includes several clients, responsible for initiating user transactions and returning the results of transaction execution to clients. In cross-shard transactions, clients are also responsible for monitoring an expired transaction or initiating transaction rollback.

The operations of DecoupleChain system mainly consists of an *initialization* stage, an *execution* stage, and a *shard reconfiguration* stage. The latter two stages are able to alternate repeatedly. A detailed explanation of these stages is as follows.

B. Initialization Stage

During the system's initialization stage, nodes in Layer2 are shuffled by invoking a common initial random source and thus allocated to different shards. Once the initial shards are constructed, the *administrator* collects the signature addresses of the validator nodes in each consensus shard. Then, the administrator deploys the smart contract `TBStore` (contract 1) in Layer1, storing these signature addresses as the contract's initial parameters. The use of these addresses is introduced in IV-E. It's essential to emphasize that the *administrator* is a centralized role that only exists during system initialization. Once the contract is deployed, the subsequent system operations no longer requires the *administrator*.

C. Transaction Execution Stage

After the system initialization is completed, the system begins to process transactions. A token-transfer transaction can be simply represented by $\texttt{tx} = \langle Sender, Recipient, Value \rangle$, where *Sender* represents the account address of the transaction payer, *Recipient* represents the payee's account address, and *Value* represents the number of tokens transferred. Let Φ denote the mapping rule from the account address to the storage shard ID. When $\Phi(Sender) = \Phi(Recipient)$, i.e., both the sender and recipient addresses are within the same storage shard, then the transaction is referred to as an *intra-shard transaction*. Otherwise, it's called a *cross-shard transaction*. We will introduce the execution process of intra-shard transactions later on, and the execution process of cross-shard transactions is elaborated on in Section V.

Let's consider the handling of an intra-shard transaction denoted as $\texttt{tx}_{\text{intra}} = \langle S_1, R_1, V_1 \rangle$. The storage shard with ID $\Phi(S_1)$, is labeled as $\texttt{SShard}_1$, and the consensus shard is labeled as $\texttt{CShard}_1$. The process from initiation to confirmation mainly includes the following phases.

- **Broadcast Phase 1**. Clients submit/broadcast transactions, including $\texttt{tx}_{\text{intra}}$, to the entire network. $\texttt{CShard}_1$ filters out $\texttt{tx}_{\text{intra}}$ from these transactions and places it in the transaction pool, shown as step ① in Fig. 3.
- **Query Phase**. After queueing for some time, $\texttt{tx}_{\text{intra}}$ is selected from the transaction pool. $\texttt{CShard}_1$ then sends a `getState` request to $\texttt{SShard}_1$, inquiring about the latest state of the accounts related to the current transaction. Upon receiving the request, $\texttt{SShard}_1$ fetches the respective account state from its local state tree, along with the Merkle Proof that confirms the validity of this state. Both the account state and the Merkle Proof are then sent to $\texttt{CShard}_1$, shown as step ③ in Fig. 3.
- **Verification and Consensus Phase**. Upon receiving the account state and the accompanying Merkle Proof from $\texttt{SShard}_1$, $\texttt{CShard}_1$ validates its accuracy in conjunction with the *time beacon* (described in the next section) stored locally. Concurrently, $\texttt{CShard}_1$ verifies the signature of each transaction. Once the verification is confirmed as correct, $\texttt{CShard}_1$ processes the transaction and packages it into a block, denoted as β. This

block then undergoes PBFT consensus, shown as the operation step ④ in Fig. 3. Upon executing the transaction, the block β contains the new state tree root hash, termed as newTrieHash. Utilizing the properties of MPT [15], $\texttt{CShard}_1$, with just the received account's Merkle Proof and the updates to these accounts, can derive the newTrieHash. $\texttt{CShard}_1$ sends β to $\texttt{SShard}_1$, and sends the receipt for $\texttt{tx}_{\text{intra}}$ to the client who initiates $\texttt{tx}_{\text{intra}}$.

- **Multi-signature Phase**. After consensus is reached, $\texttt{CShard}_1$ internally selects several signers through the VRF [23] algorithm to multi-sign the time beacon of β. A block's time beacon includes its meta-information, such as block height, shard ID, block hash, state tree root hash, transaction tree root hash, and receipt tree hash. The time beacon obtained after multi-signing is denoted as $\texttt{tb}_\beta$. The operations in this phase are shown as steps ⑤ and ⑥ in Fig. 3.
- **Broadcast Phase 2**. $\texttt{CShard}_1$, using the light node operating in Layer1, initiates a transaction (denoted as T_β) in the Layer1 chain to call the `TBStore` contract. The content of T_β is to store $\texttt{tb}_\beta$ in that smart contract.
- **Confirmation Phase**. Under our two-layer blockchain design, the Layer1 chain provides a time reference for Layer2. Suppose that T_β is packaged into block B in the Layer1 chain. Only when B is confirmed in Layer1, does T_β get confirmed. Subsequently, both β and $\texttt{tx}_{\text{intra}}$ are confirmed. When the light node in Layer1 monitors the confirmation of T_β, it immediately sends a confirmation message $\texttt{Confirm}_\beta$ to Layer2, shown as the operation step ⑧ in Fig. 3. The structure of $\texttt{Confirm}_\beta$ can be simplified as $\langle \texttt{tb}_\beta, \textit{confirmHeight} \rangle$, where *confirmHeight* refers to the height of the block in Layer1 in which T_β is packaged. Note that, the operator of each Layer2 node can either deploy a light node on Layer1 or rely on trusted, established third-party nodes. Thus, they can check if the block is confirmed in Layer1. Upon receiving $\texttt{Confirm}_\beta$, the client in Layer2 can use the transaction tree root hash and the receipt tree root hash contained within to validate the receipt of $\texttt{tx}_{\text{intra}}$, utilizing the Merkle Proof. Once the validation passes, the client returns the execution result of $\texttt{tx}_{\text{intra}}$ to the end-users. The shard nodes in Layer2, upon receiving $\texttt{Confirm}_\beta$, store it locally and use it for subsequent validations.
- **Storage and Update Phase**. When the *Verification and Consensus Phase* is completed, $\texttt{CShard}_1$ sends β to $\texttt{SShard}_1$. $\texttt{SShard}_1$ will place β in a temporary storage area, waiting for its confirmation. Upon receiving $\texttt{Confirm}_\beta$, $\texttt{SShard}_1$ first validates β, ensuring that the generated state tree root hash and other information match those in $\texttt{Confirm}_\beta$. Once validated successfully, $\texttt{SShard}_1$ adds β to the blockchain and updates the state tree accordingly, as shown in step ⑨ of Fig. 3.

D. Shard Reconfiguration Stage

After several rounds of execution, the system triggers the shard reconfiguration stage. Different from the centralized shard reconfiguration presented in Rapidchain [6] and tMPT [15], we propose a *decentralized shard reconfiguration*. When the confirmed height of the Layer1 chain reaches a certain configurable threshold since the previous reconfiguration, all consensus shards detected this change stop producing new blocks, and begin the reconfiguration process. We denote the configurable threshold as *reconfiguration interval*. We then explain the stages of the reconfiguration process. To make it easy to understand, we let $n \in \mathbb{N}^+$ represent the number of consensus shards. Thus, the consensus shard list before and after reconfiguration can be denoted by $\{cs_1, cs_2, \ldots, cs_n\}$, and $\{cs'_1, cs'_2, \ldots, cs'_n\}$, respectively. The reconfiguration process is depicted as follows.

- **Suspending consensus**. When receiving the reconfiguration signal from Layer1, shards will shut down and stop producing new blocks.
- **Determining the new consensus shards**. Taking the hash of the most recently confirmed block in the Layer1 chain as input, each node runs the VRF algorithm to obtain a verifiable random number and maps this number to an ID. This ID indicates the destination consensus shard that this node is designated. Let the shard the node was located before reconfiguration be cs_i, and the destination shard after reconfiguration be cs'_j $(1 \leq i, j \leq n)$.
- **Establishing connection**. When obtaining the VRF result, the node sends the result to its neighbor nodes, which will forward it to other nodes and consensus shards. At the same time, the node collects information about the nodes assigned to cs'_j and establishes new connections with them.
- **Synchronizing data**. cs'_j requests data from the nodes in cs_j. The primary data that needs to be synchronized is the transaction pool data of cs_j.
- **Resuming consensus**. Once the node has synchronized the data and established connections with other nodes within cs'_j, it resumes the consensus process.

E. Smart Contract `TBStore`

In this subsection, by presenting the logic of the smart contract `TBStore`, we show how Layer1 cooperates with Layer2 in our system.

The contract `TBStore` includes three publicly callable methods, namely `GetTB`, `AddTB`, and `AdjustAddrs`. The pseudocode is shown in **Contract 1**. The method `GetTB` takes two parameters, i.e., the shard ID and block height, and returns the time beacon (abbr. as `TB`) stored in the contract. The method `AddTB` verifies and stores a new time beacon. As described in IV-C, to ensure the validity of the time beacon stored in the contract, the consensus shard first needs to select several nodes through VRF (each node corresponds to a signature address). Next, the consensus shard has these nodes signing the time beacon and finally aggregates them before initiating a transaction to call the contract. Therefore, in addition to the time beacon, the parameters received by `addTB` also include the VRF results of the selected nodes and their signatures.

Smart Contract 1: `TBStore`

```
tbs : map⟨uint32, map⟨uint64, TB⟩⟩
addrs : map⟨uint32, array[address]⟩
requiredCount : uint32
Function getTB(shardID, height):
    return tbs[shardID][height]
Function addTB(tb, signatures, vrfResults, signers):
    validCount ← 0
    for i = 0 to len(signatures)-1 do
        vrf ← vrfResults[i]
        sig ← signatures[i]
        if signers[i] not in addrs[tb.shardID] then
            continue
        if ! VERIFYVRFRESULT(vrf, signers[i]) then
            continue
        if ! VERIFYSIGNATURE(sig, signers[i]) then
            continue
        validCount ← validCount+1
    if validCount ≥ requiredCount then
        tbs[tb.shardID][tb.height] ← tb
Function adjustAddrs(shardID, vrfResults, signers):
    ... // Verify vrfResults and update addrs
```

In the method `addTB`, for each signer, the contract first verifies that the signer's signature address is stored in the contract. Secondly, it checks the validity of the VRF result (i.e., whether the address is qualified as a signer for the time beacon) and then verifies the correctness of the signature. When the verification passes, the count of valid signatures is incremented. If the count of valid signatures reaches the required number, the verification passes, and the time beacon is stored in the contract. If not, the storage of the time beacon fails.

To prevent Sybil attacks [24], where a malicious node may use addresses not belonging to any consensus shard to sign invalid time beacons, the contract maintains a data structure that maps addresses to consensus shard IDs. If an attacker tries to sign a time beacon with a new address or one from a different consensus shard, the address will not pass the contract's verification process. When a consensus shard reconfiguration occurs, before submitting time beacons to the Layer1 chain, the newly formed consensus shard must update the mapping of addresses to consensus shard IDs in the contract. This is done by calling the contract's method `AdjustAddrs`. The `requiredCount` in the contract is a threshold that can be set manually to a reasonable number. When the number of valid signatures meets or exceeds the value of the threshold, there is a very low probability that the time beacon is invalid.

V. Handling Cross-shard TXs

As the global account state is divided and distributed into different shards, the transactions that their payer and payee accounts are not located in the same shard are called *cross-shard transactions*. A cross-shard transaction needs to be processed in both the payer's and payee's shards, with corresponding *debit* and *credit* operations performed. In this section, we introduce the processing mechanism of cross-shard transactions in DecoupleChain. The processing mechanism includes i) the process for successful transaction execution and ii) strategies for handling expired transactions.

A. Process of Successful Transaction Execution

We denote a cross-shard transaction as $\texttt{tx}_{\text{cross}} = \langle A, C, V \rangle$, where A is the payer account, C is the payee account, and V is the volume of tokens transferred. Denoted by $\Phi(A)$ = `src` and $\Phi(C)$ = `dest`, the successful execution of $\texttt{tx}_{\text{cross}}$ is illustrated in Fig. 4.

The execution of $\texttt{tx}_{\text{cross}}$ can be roughly divided into two phases. The first phase involves executing the debit transaction $\bar{\texttt{tx}}_{\text{cross}}$ within $\texttt{CShard}_{\texttt{src}}$ and confirming it in the Layer1 chain. The second phase involves executing the credit transaction $\hat{\texttt{tx}}_{\text{cross}}$ within $\texttt{CShard}_{\texttt{dest}}$ and confirming it in the Layer1 chain. Each phase is similar to the process of executing an intra-shard transaction. Note that, both $\bar{\texttt{tx}}_{\text{cross}}$ and $\hat{\texttt{tx}}_{\text{cross}}$ are initiated by the client, and in this process, $\texttt{CShard}_{\texttt{src}}$ and $\texttt{CShard}_{\texttt{dest}}$ do not need to communicate directly.

B. Handling Expired Transactions

The atomicity of a cross-shard transaction, say $\texttt{tx}_{\text{cross}}$, implies that either both $\bar{\texttt{tx}}_{\text{cross}}$ and $\hat{\texttt{tx}}_{\text{cross}}$ are confirmed, or they both fail to be confirmed. The cross-shard transaction $\texttt{tx}_{\text{cross}}$ is considered successful only when $\hat{\texttt{tx}}_{\text{cross}}$ is confirmed. However, due to the complexity of the system, it is possible that $\hat{\texttt{tx}}_{\text{cross}}$ still fails to be confirmed when $\bar{\texttt{tx}}_{\text{cross}}$ has been confirmed for a long time. This phenomenon threatens the execution atomicity of $\texttt{tx}_{\text{cross}}$. To better guarantee transaction atomicity, we introduce a *timeout* mechanism for dealing with cross-shard transactions. The basic idea behind such a timeout mechanism is that, if $\bar{\texttt{tx}}_{\text{cross}}$ has been confirmed and after a certain period of time, $\hat{\texttt{tx}}_{\text{cross}}$ still remains unconfirmed, the transaction is considered *expired*. When a transaction expires, there is an option to roll back $\bar{\texttt{tx}}_{\text{cross}}$. The timeout mechanism is specified as follows.

Timeout mechanism. When a client initiates $\hat{\texttt{tx}}_{\text{cross}}$, it records the confirmation height of $\texttt{CShard}_{\texttt{dest}}$ as $k \in \mathbb{N}^+$ (can be retrieved from the latest time beacon of $\texttt{CShard}_{\texttt{dest}}$) and sets a transaction **timeout timer** as $s \in \mathbb{N}^+$. If the confirmation height of $\texttt{CShard}_{\texttt{dest}}$ reaches $k + s$ and the client has not received a receipt for $\hat{\texttt{tx}}_{\text{cross}}$ from $\texttt{CShard}_{\texttt{dest}}$, the client can query $\texttt{CShard}_{\texttt{dest}}$. When $\hat{\texttt{tx}}_{\text{cross}}$ has been packed before the block at height $k + s$ (including $k + s$), $\texttt{CShard}_{\texttt{dest}}$ returns an *proof of inclusion* to the client. The client can then follow the normal process to verify the proof and complete the transaction. Otherwise, $\texttt{CShard}_{\texttt{dest}}$ returns a *proof of exclusion*, indicating that the associated transaction

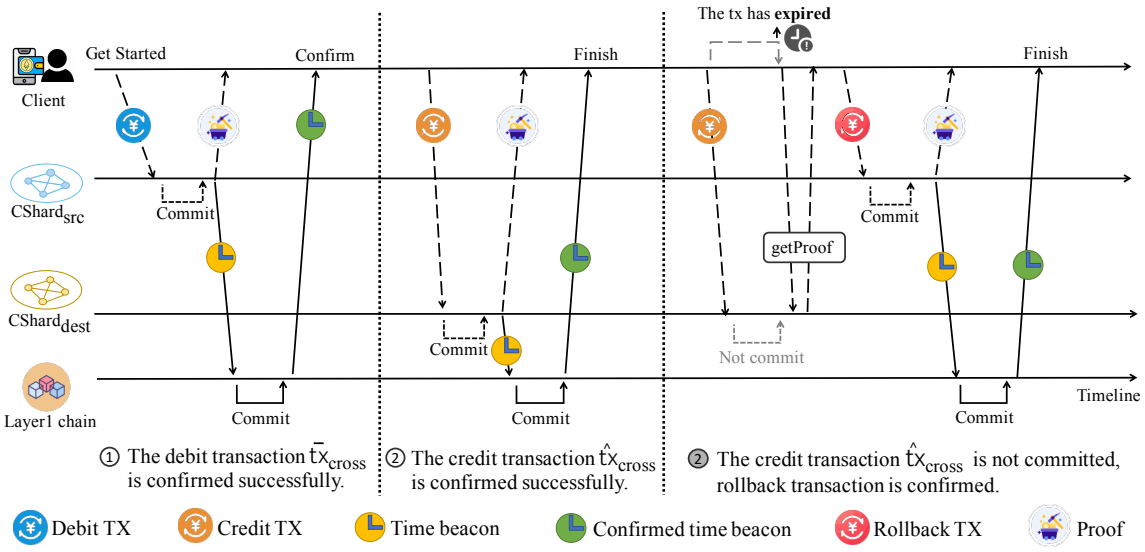


Fig. 4. The execution process of a cross-shard transaction (tx) in the proposed DecoupleChain system.

has expired. The expired transactions will be directly discarded from the transaction pool by consensus shards.

Upon receiving the *proof of exclusion*, the client can resend $\hat{tx}_{\text{cross}}$. To increase the possibility of $\hat{tx}_{\text{cross}}$ being included in a block, the client may pay a transaction fee higher than normal. Alternatively, the client can choose to initiate a *Rollback* transaction. This transaction is the inverse of $\bar{tx}_{\text{cross}}$ and aims to restore the funds to account A. Fig. 4 illustrates the rollback handling of a transaction exactly when it expires. *Rollback* transactions are prioritized by consensus shards to ensure that the transaction can be rolled back as quickly as possible.

VI. Security Analysis

A. Security of Shard Reconfiguration

When the shard reconfiguration has just been completed, the system is in a relatively secure state, where honest nodes constitute the majority within each consensus shard. To precisely measure the security of the system immediately after reconfiguration, we calculate the probability p_{fail} that at least one consensus shard may have a proportion of malicious nodes exceeding the tolerable threshold.

Assuming that the maximum tolerable proportion of malicious nodes within a consensus shard is $d \in \mathbb{R}^+, 0 < d < 1$, the proportion of malicious nodes across all nodes in the system is $\alpha \in \mathbb{R}^+, 0 < \alpha < 1$, the number of consensus shards is $S \in \mathbb{N}^+$. There are a number $n \in \mathbb{N}^+$ of nodes in each consensus shard. Then, according to [25], we have

$$p_{\text{fail}} = \frac{\left[x^{\alpha n S}\right]\left(\sum_{i=0}^{dn-1}\binom{n}{i}x^i\right)^S}{\binom{nS}{\alpha nS}}. \tag{1}$$

By setting d to 1/3 and α to 20%, we can obtain the variation of p_{fail} versus different S and n. As the number of shards increases, the size of the consensus shard needs to grow slightly to ensure that p_{fail} remains below 10^{-6}. For example, with 200 shards, a consensus shard size of 340 nodes would suffice. We argue that when this condition is met, the system achieves a sufficient level of security. Furthermore, if d decreases or α increases, this security condition can still be maintained by increasing the size of the consensus shard accordingly.

B. The Effect of Reconfiguration Interval

Even if the consensus shard is relatively secure immediately after reconfiguration, attackers can increase their voting weight in that shard by interfering or dominating the nodes within the shard through bribery attacks [9], [10]. When malicious nodes control a consensus shard, they can forge information and launch DDoS and other attacks. We leave the study of specific attack methods on consensus shards for our future work. This paper only uses a simplified attack model to analyze the effect of reconfiguration intervals.

Similar to tMPT [15], we assume the process in which honest nodes in a consensus shard turn into malicious ones follows a Poisson process. The rate $\lambda \in \mathbb{R}$ describes the average speed of nodes' corruption, i.e., honest nodes turn into malicious ones following a Poisson process with the rate λ. Considering the time required for the number of malicious nodes to grow from 0 to $n/3$, we can view the waiting time (denoted as t) for the number of malicious nodes to reach $n/3$ as the time it takes for $n/3$ independent random events to occur. According to the relationship between the Poisson process and the gamma distribution, t follows a gamma distribution with parameters $k = n/3$ and λ. Thus, the probability density function of the gamma distribution is

$$f(t;k,\lambda) = f(t;n/3,\lambda) = \frac{\lambda^{n/3}t^{(n/3)-1}e^{-\lambda t}}{(n/3-1)!}. \tag{2}$$

Thereby, the expected waiting time is

$$E(T) = \int_0^\infty t \cdot \frac{\lambda^{n/3} t^{(n/3)-1} e^{-\lambda t}}{(n/3-1)!}\, dt = \frac{n/3}{\lambda}. \quad (3)$$

We can take $T = \frac{n}{3\lambda}$ as the reconfiguration interval time (assuming we can estimate the rate λ). As λ increases, i.e., the attacker's speed becomes faster, the system needs to shorten the reconfiguration interval to maintain security. When the reconfiguration interval is reduced to a certain length, the traditional sharding protocol cannot continue due to its huge reconfiguration overhead. In contrast, our consensus shard can be reconfigured quickly and at a low cost as needed due to its stateless feature.

C. Security of Storage Shards

DecoupleChain only reshuffles the consensus shards and does not reshuffle the storage shards, which may raise some concerns. Our view is that storage shards lack the motivation to act maliciously, as there is no benefit to gain from possible malicious actions. When storage shards intentionally return incorrect account states to the consensus shards, those incorrect states will be rejected due to a failure in the verification phase. Future work will study this aspect further and propose corresponding solutions, such as the detection and replacement of malicious nodes in storage shards.

VII. Experiments

A. Prototype Implementation

To evaluate the performance of DecoupleChain, we implemented a prototype of our system using Go with around 15,000 lines of code. We then deployed the `TBStore` contract on a private Ethereum blockchain, which serves as the Layer1 chain for DecoupleChain. We collected historical blocks from Ethereum that include 1.92 million token-transfer transactions that occurred from June 7, 2022, to June 14, 2022. We then replay those transactions in our prototype system. Specifically, we first read all the accounts associated with these transactions and divide them into different shards according to the last few bits of their account addresses. Then, we inject a certain number of transactions per second into clients, simulating the users' behavior of submitting transactions in real time. When the transaction injection is complete, we read the transaction execution results from the log files.

B. Experiment Settings

Testbed. We conducted our experiments on our local 9 high-performance workstations. Each workstation is equipped with an Intel(R) Xeon(R) W-2150B CPU @ 3.00 GHz, which consists of 10 physical cores and supports 20 threads in total. Additionally, each machine is equipped with 64 Gigabytes (GB) of memory.

Basic Parameters. To set a relatively reasonable block capacity and block interval, we investigated the parameter settings of classic papers, which are shown in Table I. Reffering to their settings, we set the block interval to 4 seconds and the block capacity to 1000 transactions. Thus, the highest theoretical throughput that a single shard can achieve is 250 transactions per second (TPS). The number of nodes in each shard is initially set to 16.

TABLE I
The settings of several sharding systems and our system.

System	Monoxide [7]	BrokerChain [8]	tMPT [15]	Ours
TPS per shard	15∼20	62.5	250	250
Block interval	15s	8s	4s	4s

Baselines. We choose the following three methods as the baselines to compare with the performance of DecoupleChain. The *Full sync* (Ethereum) method [3] requires synchronizing all block data from the genesis block to the latest block, while the *Fast sync* (Ethereum) method [3] only synchronizes recent blocks and the state tree. The *tMPT* method [15] compresses the state tree based on active accounts, and nodes must synchronize the compressed state tree.

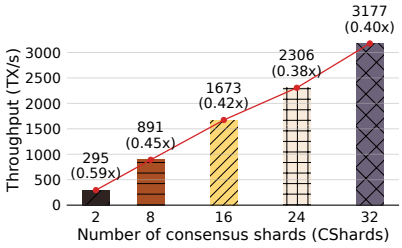
C. Overall Metrics of DecoupleChain

We first measure the two most critical performance metrics of DecoupleChain, i.e., TPS and transaction confirmation latency. The results are shown in Fig 5. TPS refers to how many transactions the system can process per second, and transaction confirmation latency refers to the time it takes for a transaction to be confirmed by the Layer1 chain since a client's submission. The experimental results in Fig. 5(a) show that as the number of consensus shards increases, the throughput increases accordingly. When we focus on the ratio of actual TPS over the theoretical maximum TPS, we observe a similar trend in Fig. 5(b) to the average latency, which can be explained as follows. As the number of shards grows, the proportion of cross-shard transactions rises, leading to increased average transaction confirmation latency and a reduced TPS ratio. However, when the shard count reaches 32, the ratio of cross-shard transactions stabilizes. At this point, the load balancing effect of multi-sharding becomes more apparent, such that the average transaction latency drops and the TPS ratio increases.

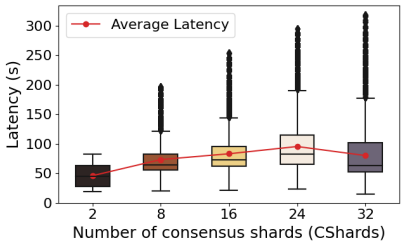
We also present the workload distribution across each consensus shard in Fig. 5(c). Note that a cross-shard transaction generates one individual unit of transaction workload in each of the two stages of its execution. In terms of workload per shard, our results are somewhat similar to those of Monoxide [7]. In Monoxide, a shard node often needs to participate in block production for multiple shards, thus bearing the workload of several shards. In contrast, in DecoupleChain, each node only handles the transaction workload of the shard where it is located.

D. How System Parameters Influence Performance

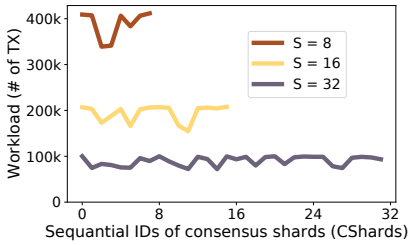
1) Impact of transaction inject speed on TPS: To evaluate the system's performance under varying transaction arrival rates, we test DecoupleChain by starting with a low transaction injection speed and gradually increasing the volume of transactions injected per second. Our empirical study shows that when the transaction injection rate is set to the maximum



(a) Increasing TPS versus the number of CShards. The coefficient in parentheses represents the ratio of TPS over the theoretical maximum TPS.

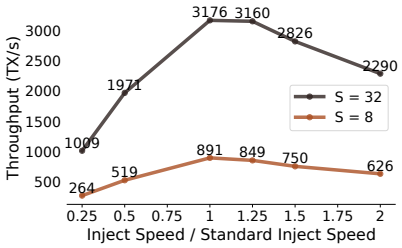


(b) Distribution of transaction confirmation latency versus different numbers of CShards.

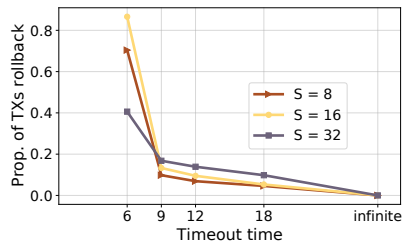


(c) Distribution of transaction workloads across CShards.

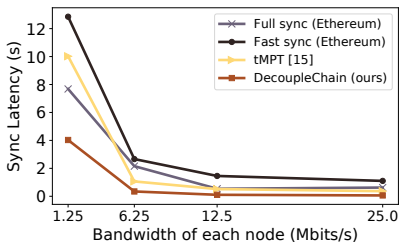
Fig. 5. Throughput, transaction confirmation latency, and transaction workload distribution of the DecoupleChain system.



(a) Impact of transaction injection speed on throughput.

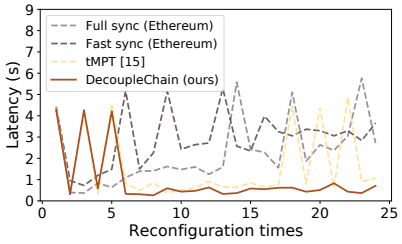


(b) Proportion of transactions rolled back because of failing to commit before timeout. Here, "infinite" means that transactions will never expire.

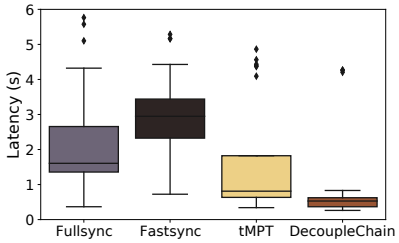


(c) The data synchronization latency during reconfiguration under various bandwidth settings.

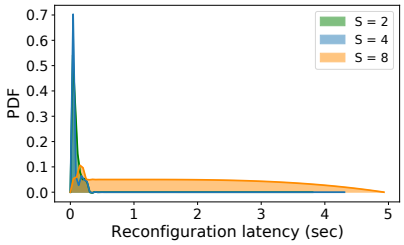
Fig. 6. Effects of various system parameters on the performance of throughput, transactions rolled back, and the data synchronization latency.



(a) The reconfiguration latency fluctuates over time under different methods.



(b) The reconfiguration latency distribution of different methods.



(c) Probability distribution function (PDF) of reconfig. latency while varying the # of CShards.

Fig. 7. Shard reconfiguration latency.

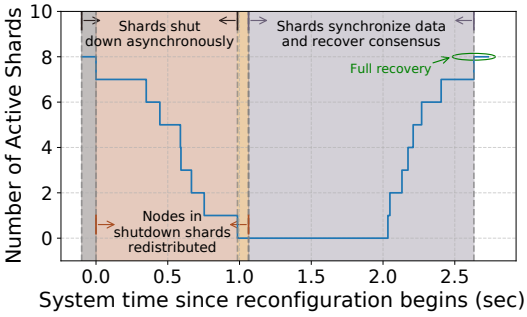


Fig. 8. Asynchronous reconfiguration process under *Full sync* method.

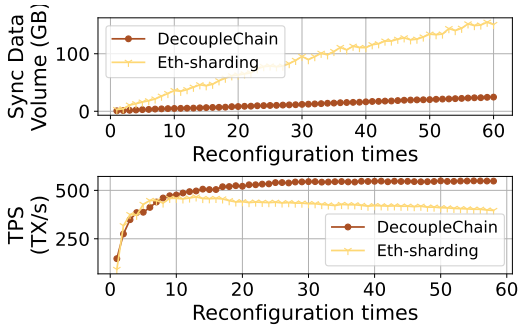


Fig. 9. Comparing with *Eth-sharding* under frequent shard reconfigurations.

transaction processing capacity of the entire system (i.e., the maximum theoretical TPS of a single shard multiplied by the number of shards), the system can achieve a TPS close to the optimal level. We refer to this injection rate as the *Standard Injection Speed*. The results are shown in Fig. 6(a). When the transaction injection rate is either too low or too high,

the system's throughput cannot achieve the best. However, when the condition of *Standard Injection Speed* is met, the throughput of DecoupleChain shows the best.

2) Impact of timeout timer on rollback rate: To provide a reference for setting the timeout timer of transactions (defined in V-B), we study the variation in the transaction rollback rate when different timeout lengths of timers are set. When a transaction expires, the client initiates a rollback transaction. The timeout timer's unit is the block interval of the consensus shard. Let $r \in \mathbb{R}$ $(0 < r < 1)$ denote the transaction rollback rate, $a \in \mathbb{N}^+$ the number of successfully executed transactions, and $b \in \mathbb{N}^+$ the number of successfully rolled-back transactions, then $r = \frac{b}{a+b}$. The results are shown in Fig. 6(b). The observation shows that a longer timeout timer induces a lower transaction rollback rate, which is consistent with our expectations. Clients can flexibly adjust the transaction timeout timer according to customized needs and choose a relatively reasonable range (e.g., 12–18 or more than 18 block intervals).

3) Impact of bandwidth on synchronization latency: We compare DecoupleChain to three other baselines (i.e., *Full sync* [3], *Fast sync* [3], and tMPT [15]), focusing on the data synchronization phase during shard reconfiguration, which represents the key improvement of DecoupleChain. The results in Fig. 6(c) show that under different bandwidth settings, DecoupleChain achieves the lowest synchronization latency during reconfiguration. This is because during shard reconfiguration, nodes in DecoupleChain only need to synchronize the transaction pool data, whereas the other three methods require the synchronization of additional content in addition to the transaction pool. In the following experiments, we set the bandwidth of each node to 12.5 Mbits/s and measure the latency of the entire configuration process.

E. Reconfiguration Latency

We compare the shard reconfiguration latency of different methods under the same settings in Fig. 7(a) and Fig. 7(b). DecoupleChain achieves the shortest overall reconfiguration latency, the median reconfiguration latency of DecoupleChain is 35% lower than tMPT and 67% lower than *Fast sync*. The large fluctuations in Fig. 7(a) are primarily due to significant variations in the latency of the shutdown phase during each shard reconfiguration explained in next subsection. Additionally, since the accumulated blockchain data can increase over time, the latency of the *Full sync* and *Fast sync* methods will increase as the data size grows, while the *tMPT* method and DecoupleChain remain relatively stable. We also measured the impact of different numbers of shards deployed on the testbed machines. The results in Fig. 7(c) show that the reconfiguration latency increases slightly as the number of consensus shards grows. This is mainly because the shutdown duration becomes longer as more shards exist in the system.

F. Zooming in Reconfiguration Process

To help better understand the large fluctuations of reconfiguration latency shown in Fig. 7(a), we use *Full sync* [3] as the data synchronization method to zoom in the shard reconfiguration process. In Fig. 8, we see that during this process, some consensus shards shut down faster than that under other shards because other shards may still be generating a new block or multi-signing a time beacon, leading to a dynamic decrease in the number of active shards. When recovering consensus under *Full sync*, some shards synchronize data and rebuild connections more quickly than those slow shards. In our settings, the shutdown phase does not exceed one block interval (i.e., 4 seconds) by too much. Since the redistributing phase can be pipelined with the shutdown phase, it will be completed shortly after the shutdown phase. The latency of the recovery phase depends on the synchronization method used and the volume of data.

G. Comparison with Other Sharding System

To provide stronger evidence that DecoupleChain outperforms other sharding blockchains, we also implemented a prototype of the sharding system on top of Ethereum (shortened as Eth-sharding) and compared its reconfiguration data volume and throughput with DecoupleChain. We set the number of shards to 4 for each system, with 16 nodes for each shard, and triggered reconfiguration every three block intervals (around 12 seconds). The results shown in Fig. 9 demonstrate that DecoupleChain has a lower volume of synchronization data during reconfiguration compared to Eth-sharding. As the system continues to run and the data size increases, the advantages of DecoupleChain become more obvious. Furthermore, while Eth-sharding's throughput is significantly influenced by frequent reconfigurations, DecoupleChain remains stable TPS.

VIII. Conclusion

DecoupleChain is a two-layer blockchain sharding system that decouples traditional sharding consensus from storage, enabling frequent reconfigurations and thereby enhancing the security of the blockchain sharding system. DecoupleChain delegates the functionalities of storage and consensus into different shards. To ensure the atomicity of transaction execution, we proposed a correctness-verification method for validating accounts' state and transaction receipts, as well as a dedicated timeout mechanism. Experimental results demonstrate that DecoupleChain not only outperforms existing sharding systems in terms of reconfiguration overhead but also maintains high performance even under frequent shard reconfigurations.

In our future work, we plan to make DecoupleChain a real-world infrastructure for future public blockchains.

Acknowledgment

This work was partially supported by the National Key R&D Program of China (No. 2022YFB2702304), GuangDong Basic and Applied Basic Research Foundation (2025B1515020053), and NSFC (No. 62272496).

REFERENCES

[1] V. Buterin, "Why sharding is great: demystifying the technical properties," *Vitalik. ca, available at: https://vitalik. ca/general/2021/04/07/sharding.html*, 2021.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, pp. 1–8, 2008.

[3] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A Secure Sharding Protocol For Open Blockchains," in *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, 2016, pp. 17–30.

[5] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *Proc. of IEEE Symposium on Security and Privacy (SP'18)*, 2018, pp. 583–598.

[6] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, 2018, pp. 931–948.

[7] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *Proc. of 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, 2019, pp. 95–112.

[8] H. Huang, X. Peng, J. Zhan, S. Zhang, Y. Lin, Z. Zheng, and S. Guo, "Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding," in *Proc. of IEEE Conference on Computer Communications (INFOCOM'22)*, 2022, pp. 1–10.

[9] S. Gao, Z. Li, Z. Peng, and B. Xiao, "Power adjusting and bribery racing: Novel mining attacks in the bitcoin system," in *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS'19')*, 2019, pp. 833–850.

[10] H. Sun, N. Ruan, and C. Su, "How to model the bribery attack: A practical quantification method in blockchain," in *Proc. of the 25th European Symposium on Research in Computer Security (ESORICS'2020)*. Springer, 2020, pp. 569–589.

[11] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, vol. 99, 1999, pp. 173–186.

[12] C. Li, P. Li, D. Zhou, Z. Yang, M. Wu, G. Yang, W. Xu, F. Long, and A. C.-C. Yao, "A decentralized blockchain with high throughput and fast confirmation," in *Proc. of USENIX Annual Technical Conference (ATC'20)*, 2020, pp. 515–528.

[13] A. Mariani, G. Mariani, D. Pennino, and M. Pizzonia, "Blockchain scalability and security: Communications among fast-changing committees made simple," in *Proc. of the 20th International Conference on Software Architecture Companion (ICSA-C'23)*, 2023, pp. 209–215.

[14] D. Tennakoon, Y. Hua, and V. Gramoli, "Smart red belly blockchain: Reducing congestion for web3," in *Proc. of the 37th IEEE International Parallel & Distributed Processing Symposium (IPDPS'23)*, 2023.

[15] H. Huang, Y. Zhao, and Z. Zheng, "tmpt: Reconfiguration across blockchain shards via trimmed merkle patricia trie," in *Proc. of the 31st International Symposium on Quality of Service (IWQoS'23)*, 2023, pp. 1–10.

[16] M. Li, Y. Lin, J. Zhang, and W. Wang, "Cochain: High concurrency blockchain sharding via consensus on consensus," in *Proc. of IEEE International Conference on Computer Communications (INFOCOM'23)*, 2023, pp. 1–10.

[17] A. Rauch, "Student research abstract: Splitchain: Blockchain with fully decentralized dynamic sharding resilient to fast adaptive adversaries," in *Proc. of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC'23)*, 2023, pp. 280–283.

[18] M. Xu, Y. Guo, C. Liu, Q. Hu, D. Yu, Z. Xiong, D. Niyato, and X. Cheng, "Exploring blockchain technology through a modular lens: A survey," *arXiv preprint arXiv:2304.08283*, 2023.

[19] M. Li, Y. Lin, J. Zhang, and W. Wang, "Jenga: Orchestrating smart contracts in sharding-based blockchain for efficient processing," in *Proc. of the 42nd IEEE International Conference on Distributed Computing Systems (ICDCS'22)*, 2022, pp. 133–143.

[20] M. Xu, Q. Wang, H. Sun, J. Lin, and H. Huang, "W3chain: A layer2 blockchain defeating the scalability trilemma," in *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2023, pp. 1–9.

[21] A. Yakovenko, "Solana: A new architecture for a high performance blockchain v0. 8.13," *Solana Whitepaper*, 2018.

[22] Ethereum team, "Optimistic rollups," https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/, 2021.

[23] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *Proc. of 40th annual symposium on foundations of computer science (FOCS'99)*, 1999, pp. 120–130.

[24] J. R. Douceur, "The sybil attack," in *International workshop on peer-to-peer systems*, 2002, pp. 251–260.

[25] A. Hafid, A. S. Hafid, and M. Samih, "A tractable probabilistic approach to analyze sybil attacks in sharding-based blockchain protocols," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 1, pp. 126–136, 2022.