Hierarchical Execution of Cross-Shard Transactions with Multiple Smart Contract Functions

Abstract— **A sharding framework has been proposed by Ethereum 2.0, and researchers have tried to enhance its applicability and scalability to real blockchain networks. Each shard can execute transactions requested by users, so the number of transactions dealt with by the shard-based blockchain grows as the number of shards increases. Dealing with cross-shard transactions, however, is a major hindrance to blockchain performance, because each such transaction requires cooperation among different shard validators in the network. Given this background, this paper proposes a novel cross-shard architecture in which smart contract functions in different shards are called in a hierarchical manner so as to dramatically reduce the interactions among different shard validators. We describe the proposed cross-shard framework in detail and demonstrate how a cross-shard transaction can be executed using the hierarchically connected smart contract functions. In the proposed architecture, the interactions among different shard validators can be reduced by having each validator select its neighbors from among members of its own shard after each epoch. This frequent change of neighbors for each validator makes it robust against eclipse attacks. In addition, this architecture speeds up the validation process of transactions/blocks in each shard; we show the effect by measuring the block transmission delays in the proposed architecture in a simulation environment.**

Keywords—blockchain, Ethereum, shard, cross-shard transaction, neighbor selection

I. Introduction

Blockchain, which was originally proposed for transfer of Bitcoin between online users [1], has now become the platform of the new World Wide Web, called Web 3. Decentralized applications (DApps) including non-fungible token (NFT) trading, decentralized finance (DeFi), gaming, and social media are now run on Web 3, and their users are growing in number. In Q2 2023, the number of daily unique active wallets (dUAWs) connected to DApps increased by 7.97% from the previous quarter and surpassed 1,940,000 [2]. Such a huge number of transaction requests from users creates congestion on current smart contract blockchains, such as Ethereum [3].

The Ethereum 2.0 sharding architecture has recently been proposed as a way to tackle this congestion problem [4]. In this architecture, each shard deals with transactions requested from users in parallel, so the number of transactions dealt with in an epoch duration is expected to grow as the number of shards is increased. Moreover, dynamic blockchain sharding [5] has been proposed, wherein each shard is dedicated to one DApp so that transaction requests sent to each DApp are executed by the shard independently without considering the other shards. Dynamic blockchain sharding, however, does not consider cross-shard transactions, which span multiple DApps; thus, its usage is restricted to transactions in a single shard.

On the other hand, there are other forms of sharding, such as ChainSpace and OmniLedger [7], that do consider cross-shard transactions. Their cross-shard transactions, however, require each user to specify the shards involved in the requested transaction and the user also has to manage a two-phase commit for the transaction. In addition, the validators in the involved shards have to multicast the validation results to each other to confirm the transaction results in other shards. Therefore, validators cannot concentrate on the transactions executed in their own shards.

For this circumstance, we propose a sharding architecture in which the smart contract function called by a user conducts the two-phase commit for the cross-shard transaction in a hierarchical manner by cooperating with functions in other shards. By using the proposed sharding, users don't have to specify the shards involved in each transaction, and validators don't have to multicast their validation results to other shards; thus, they can focus on validations of the transactions in their own shards.

In summary, this paper makes following contributions.

- In the proposed sharding, each validator in each shard replaces its neighbors whenever its shard members are changed after each epoch, so that its neighbors are always selected from the same shard. This neighbor-selection method enables each validator to focus on validating transactions executed in its shard, and it significantly reduces communications between different shard validators. In addition, this frequent change of neighbors for each validator helps to reduce the risk of eclipse attacks [8] [9].
- In the proposed sharding, the smart contract function receiving a cross-shard transaction requested from a user calls functions in other shards, and these called functions can recursively call functions in other shards in a hierarchical manner. In this way, a variety of different cross-shard transactions can be created by combining functions in different shards. The hierarchically top function can manage the atomicity and consistency of a cross-shard transaction, so the two-phase commit of the transaction is assured. Thus, there is no need for the user to specify the shards involved in a cross-shard transaction.
- The effectiveness of the proposed sharding is quantitatively evaluated in terms of the transmission delays of a block from a source validator to the other validators in the same shard. In this evaluation, not only the propagation delays but also queuing delays are considered. As a result, it is found that the proposed sharding shortens these delays by forcing each validator to set its neighbors to be ones within its own shard and by

removing the burden of forwarding blocks for the other shards.

The remainder of this paper is as follows. Section II summarizes related work, while Section III presents background technology for this study. Section IV shows the proposed sharding architecture and an application example. Section V describes the evaluation results using a simulator and discusses the applicability of the proposed sharding. Section VI concludes this paper.

II. Related Work

There are two different blockchain transaction models; the UTXO (unspent transaction output) model represented by Bitcoin and the account/balance model represented by Ethereum. OmniLedger [7] and RapidChain [10] are the major proposed shardings for the UTXO model; they partition validators into multiple shards to increase the throughput (executed transactions/sec) in the blockchain. However, in the UTXO model, it is inevitable that different shard validators will have to communicate to get the whole history for each user transaction. In addition, smart contracts are generally not compatible with the UTXO model, so the model is out of the scope of the proposed sharding.

We apply the proposed sharding to the account/balance model. Besides the ones mentioned in the previous section, there are other shardings that use this model. D-GAS (dynamic load balancing mechanism among Ethereum shards) [11] focuses on the Ethereum default setting in which each user account is allocated to a shard on the basis of the account address, meaning that loads are generally different among the shards. Thus, D-GAS predicts the future transaction volume of each shard and reallocates the user accounts among different shards, so that each shard will receive a balanced transaction load. D-GAS, however, does not consider cross-shard transactions.

Okanami et al. proposed a method of balancing loads among shards that considers cross-shard transactions [12]. In this method, some nodes in the blockchain play the role of competition coordinator and ask third parties to compete for providing the best load-balancing solution among shards by giving the third parties past shard-load information. The third parties return the coordinators their solutions as to how the users and smart contract accounts are assigned to individual shards for a future epoch, and the winner of the competition can get a reward from the coordinators. However, this method is costly because it has to pay a reward in each epoch, and it does not consider how to alleviate the burden of cross-shard transactions on individual validators.

Hong et al. proposed to apply a distributed database to a sharding blockchain [13]. In that proposal, each table of the distributed database is allocated to one of the shards and the table is updated or queried by the shard on the basis of the transaction data verified in the shard. A blockchain user, however, may send a cross-shard transaction accessing different tables located in multiple shards. In this case, the main shard, which is responsible for conducting the transaction, has to download the necessary tables from the other shards; thus, it imposes a large burden on the shard. Thus, the sharding we propose does not consider distributed database mapping with all the shards. Instead, it assumes each validator holds the transactions verified by the shard in its own storage or validators in a shard share a distributed database, which is independent from the databases in other shards.

There have been many studies on neighbor selection for each validator/node in a blockchain network, because fast propagation of a transaction/block to other nodes in a blockchain network is important for enhancing transaction throughput. In a blockchain with the PoW (proof of work) consensus algorithm, the longer it takes to propagate a block to all the nodes in the network, the higher the fork rate of the blockchain in the network will be [14]. In addition, Kertesz et al. showed that having about eight neighbors for each node helps to minimize the fork rates in networks of various sizes [15].

In the default configuration, each Bitcoin node picks eight nodes randomly as its outbound neighbors [16], but this random choice may be inappropriate for fast transaction/block propagation. Thus, many studies have tried to shorten the block propagation time in a blockchain network by periodically replacing the neighbors of each node with nodes that have faster block propagation speeds [17], wider bandwidth [18], or shorter physical distances [19].

Baniata et al. proposed a minimum spanning tree (MST)-based neighbor-selection method [20] in which a leader creates an MST considering all the network edge costs and each node selects its neighbors based on the created MST. Matsuura et al. proposed a region-based neighbor-selection method [21] in which each node selects more neighbors from its own region, such as six out of eight neighbors, to speed up block propagation. All of these methods are, however, aimed at a blockchain network without considering shard validator allocations. In other words, even if some shard validators are moved from one shard to others, the neighbors of each validator are not changed.

III. Research Background

This section reviews the basics of blockchain technology, including the sharding framework proposed in Ethereum 2.0, dynamic blockchain sharding, and the gossip protocol [22] that is applied to transaction/block propagation in a blockchain network.

A. Ethereum 2.0 sharding framework

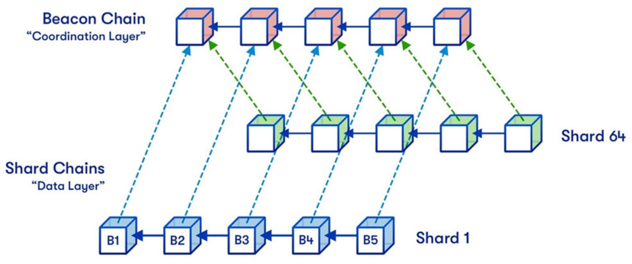


Fig. 1. Ethereum 2.0 sharding framework.

Figure 1 shows the Ethereum 2.0 sharding framework described in [23]. In this example, 64 shard chains are coordinated by a Beacon chain and each Beacon chain block manages individual blocks in the shards, as indicated by the dotted arrows. In this way, transactions from users are parallelly executed by multiple shards and their transaction results are stored in the blocks in the corresponding shards. The Beacon chain manages the created shard blocks by linking them to its corresponding blocks.

B. *Dynamic blockchain sharding*

In dynamic blockchain sharding, the Beacon chain allocates a shard to each DApp and it also replaces some validators in each shard after each epoch in order to prevent shard takeover attacks from adaptive adversaries [24]. Different from other shardings, not only validators but also the number of shards and the upper limit of validators for each shard are dynamically changed in accordance with the judgement of the Beacon chain [5].

Figure 2 shows an application example of dynamic blockchain sharding. The validators in each shard are replaced over time (epoch interval). Three shards are created in Epoch 1, and each shard has nine validators, which are set by the admin nodes in the Beacon chain, and the nodes that have expressed their intention to join one of the shards are randomly assigned to each shard. Each node receives from the Beacon chain the shard tag of the one it will join and a list of IP addresses of all nodes participating in the same shard, so that each node can become a validator in the shard and connect with other validators belonging to it.

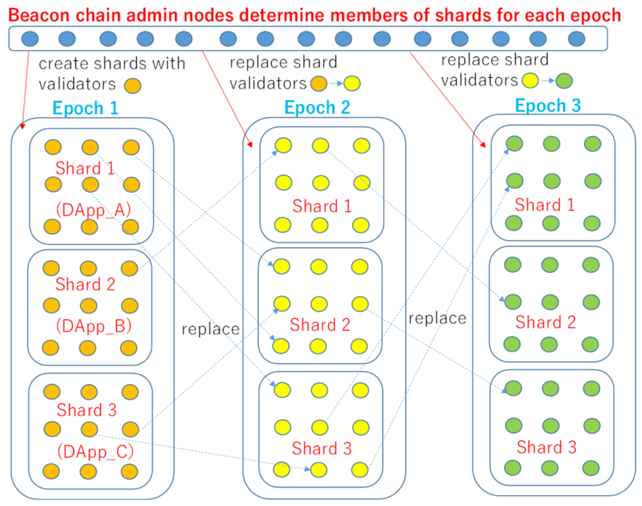


Fig. 2. Dynamic blockchain sharding example.

Before each epoch starts, each node again receives a new shard tag and its shard-member IP addresses from the Beacon chain on the basis of the judgement of the admin nodes. Therefore, these shard validators are shuffled among the shards, so it is difficult for an adversary to take over any one shard. The proposed sharding is assumed to be run within this dynamic blockchain sharding framework.

C. *Gossip protocol used for transaction/block propagation*

Transaction/block propagation in a blockchain network is conducted by the gossip protocol. The gossip protocol defines rules by which participants in a distributed system exchange information probabilistically [22]. In a blockchain network, each node sends a transaction/block to its outbound neighbors and each neighbor that receives a transaction/block forwards it to its own neighbors unless the neighbor has already received the same transaction/block. Outbound neighbors are destination nodes, to which each node sends a transaction/block directly in one hop. On the other hand, each node also has inbound neighbors in addition to its outbound neighbors. An inbound neighbor node is the source node from which the node receives a transaction/block directly in one hop.

Transaction/block propagation within each shard is also performed using the gossip protocol [25], because each shard belongs to a blockchain network. In the existing shardings, however, each validator can have some outbound/inbound neighbor validators allocated to other shards. In that case, each validator has to forward transactions/blocks to validators in other shards, even though they are not used in its shard. If a validator does not forward unnecessary transactions/blocks that are not used by its shard, its outbound neighbors cannot receive them, though some of them are necessary. Therefore, the transaction/block transfer load of each validator is increased in proportion to the number of shards. Furthermore, since each transaction/block is passed among all validators in all the shards, it is inevitable that the propagation time of a transaction/block in each shard will be long.

To tackle these problems, the proposed sharding confines the neighbors of each validator to be within it. Instead, it has smart contract functions in different shards hierarchically cooperate with other and manage the corresponding cross-shard transaction. This feature makes it possible to execute cross-shard transactions without using the gossip protocol between different shards.

IV. Proposed Sharding Enabling Hierarchical Cross-Shard Transactions with Smart Contracts

This section describes the proposed sharding framework. It also describes the proposed hierarchical two-phase commit for a cross-shard transaction among different shards through their smart contract functions.

A. *Proposed sharding framework*

The proposed sharding is assumed to be run within the dynamic blockchain sharding framework; thus, each shard is allocated to a DApp and shard validators are decided in accordance with the judgement of the Beacon chain at the end of each epoch. The proposed sharding is, however, different from the dynamic blockchain sharding in terms of its neighbor selection, in which each shard validator selects its neighbors in the shard and changes neighbors after each epoch in a way that neighbors are restricted to be within its shard, as well as that it deals with cross-shard transactions.

Figure 3 illustrates an example of the neighbor-selection process of the proposed sharding. The neighbor-node selection and replacement are based on the ShardNodes event sent from the Beacon chain to the shard nodes. The Beacon-chain smart contract has the Join function, which is called by a new node running for a shard validator. When the number of shard validator candidates reaches a specified number, the Create function is called at the discretion of the Beacon chain, and a shard is generated as a result. The tag for the created shard, the IP address list of the shard validators and the wallet account list of the shard validators are sent to the validators participating in the shard.

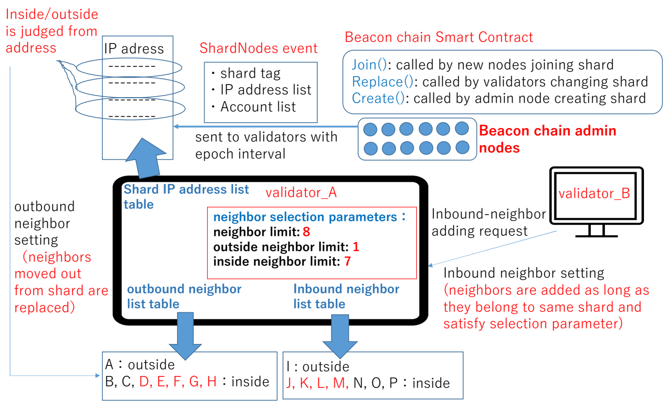

Inside/outside is judged from address
IP adress
ShardNodes event
・shard tag
・IP address list
・Account list
Beacon chain Smart Contract
Join(): called by new nodes joining shard
Replace(): called by validators changing shard
Create(): called by admin node creating shard
Beacon chain admin nodes
sent to validators with epoch interval
Shard IP address list table
validator_A
neighbor selection parameters :
neighbor limit: 8
outside neighbor limit: 1
inside neighbor limit: 7
outbound neighbor list table
Inbound neighbor list table
validator_B
Inbound-neighbor adding request
Inbound neighbor setting
(neighbors are added as long as they belong to same shard and satisfy selection parameter)
outbound neighbor setting
(neighbors moved out from shard are replaced)
A : outside
B, C, D, E, F, G, H : inside
I : outside
J, K, L, M, N, O, P : inside

Fig. 3. Setting the neighbors of each validator in its shard in the proposed sharding.

In this example, the proposed sharding uses the region-based neighbor-selection method [21], in which each validator selects fewer neighbors from outside the region (these are called outside neighbors, while those selected from the same region are called inside neighbors). Here, each validator has at most one outside neighbor and at most seven inside neighbors for its inbound and outbound neighbors, and it selects them from those within its own shard. Thus, it may have to replace some of its neighbors in the epoch interval. That is why we chose the region-based method; i.e., it uses the IP address of each shard node given by the Beacon chain to swiftly choose inside and outside neighbors.

In the illustration, when validator_A joins a shard for the first time, it receives a ShardNodes event from the Beacon chain, and it chooses eight validators from the shard's IP address list table as outbound neighbors and sets them in the outbound neighbor list table. The maximum numbers of inside and outside neighbors are seven and one, respectively. Validator_A also sets its inbound neighbors in response to other validators' inbound-neighbor adding requests, as long as the requester is located in the same shard and its inside and outside neighbors are within the maximum limits.

Honest validators in each shard call the smart contract's Replace function to avoid fixing shard members to one shard in order to protect the shard from malicious attacks. When the number of calls to the Replace function reaches a threshold, the smart contract's Create function is called again to replace the shard member, and a new ShardNodes event is sent to the participating nodes of the shard. Upon receiving the event, each shard validator can update the IP address list of its shard member.

As a result, in the outbound neighbor list table, neighbors that no longer exist in the shard are replaced with validators belonging to the same shard. In addition, neighbors that no longer exist in the shard are deleted from the inbound neighbor list table, and inbound-neighbor adding requests from other validators are accepted again up to the maximum limit of inside and outside neighbors.

For example, in Fig. 3, the outbound neighbor list table shown in red has five validators (D, E, F, G, H), and the inbound neighbor list table has four validators (J, K, L, M). If these nine nodes are not included in the IP address list in the new ShardNodes event sent from the Beacon chain, it means that they have left the shard to which validator_A belongs. In this case, validator_A replaces (D, E, F, G, H) with five randomly selected inside nodes from the shard IP address list. In addition, the four validators (J, K, L, M) in the inbound neighbor list table are replaced with the nodes sending the inbound-neighbor adding request. In this process, validator_A checks whether the address of the requesting node is included in the shard IP address list table and the node must be located inside the region because all four validators (J, K, L, M) are inside neighbors.

In this way, each validator chooses only neighbors in its own shard, so there is no need to forward transactions/blocks to other shards. In addition, since it is not necessary to pass through other shard nodes when propagating transactions/blocks within each shard, it is expected that the transaction/block propagation time for each shard will be shortened. However, we still have to consider cross-shard transactions that so far have required multicast communication among different shard validators. Thus, in the following subsections, we will show how cross-shard transactions are conducted in the proposed sharding.

B. *Cross-shard transaction using hierarchical cooperation among smart contract functions*

In the previous shardings, the user needs to specify all the shards involved in a cross-shard transaction so that data atomicity is assured, and each cross-shard transaction involves a lot of data exchanges between validators in different shards. This subsection describes a hierarchical cross-shard transaction management that solves these problems, in which a smart contract function called by a user recursively calls shard smart contract functions hierarchically, and the top function also manages the two-phase commit of the cross-shard transaction.

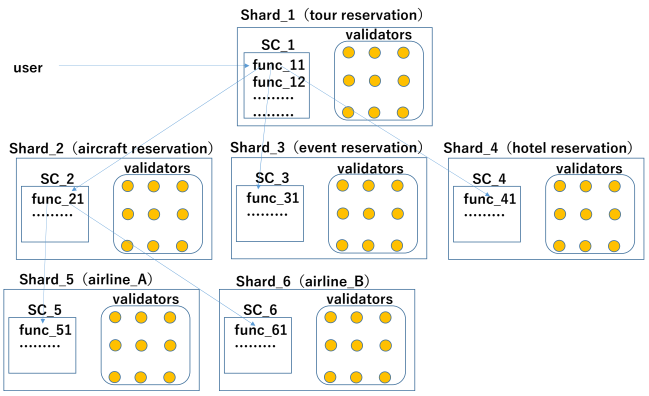


Fig. 4. Hierarchical cross-shard transaction execution example managed by smart contract functions.

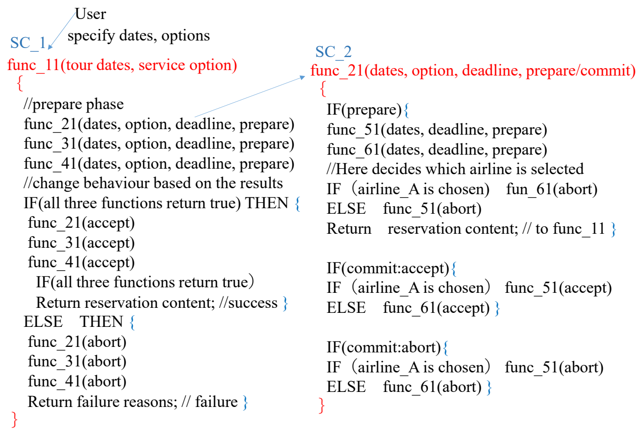
Figure 4 shows an example how the proposed hierarchical management is conducted through cooperation of the smart contract functions in the related shards. Here, Shard_1 is for a tour reservation service, while SC_1 is a smart contract located in Shard 1, and func_11 is the function that returns to the user a notification of whether the tour has been successfully booked on the specified dates.

Shard_2 is for an airline reservation application, and SC_2 is a smart contract in Shard 2. The function, func_21, checks whether an airline reservation can be obtained on the specified dates and returns the result. Each airline has its own shard; Shard_5 is for airline_A and Shard_6 is for airline_B, and each of them has smart contracts SC_5 and SC_6, respectively. Func_51 or func_61 checks the availability of each airline on the designated dates. If there are vacancies in both companies, func_21 reserves one by making a comparison taking the preferences given by the user into consideration. Then, it returns the result to func_11.

Shard_3 is for an event reservation application. SC_3 reserves events desired by users depending on the place and date, and func_31 makes an event ticket reservation for the specified date. Shard_4 is for a hotel reservation application. SC_4 conducts hotel reservations based on travel destinations, and func_41 makes a hotel reservation on the specified dates.

In this way, a hierarchical relationship is created at the function level among different shard smart contracts, and each function performs a two-phase commit management only for the functions one level below it, which balances the load among the functions. In addition, if the hierarchically top function, func_11, is notified that the reservation process in a function under it has failed, the locked data in all the related shards for the cross-shard transaction can be rolled back all at once in the commit phase, so that the atomicity and consistency of the data among multiple shards is maintained.

Figure 5 shows an example of how func_11 and func_21 in Fig. 5 are implemented. Func_11 is the hierarchically top function conducting the cross-shard transaction requested by the user in Fig. 5. Func_21 is one level down in the hierarchy from func_11 as are func_31 and func_41, and they are called by func_11, as shown in the pseudo code. When func_11 calls func_21, func_31, or func_41 in the prepare phase, a reservation is attempted for the dates specified by the user. At that time, the reply deadline for each function can be specified as an argument of the function, and the reservation request is canceled when this deadline expires. These deadline time periods may be set differently for each function to be called, and the unit of time (days, hours, minutes, seconds, etc.) can also be determined by the policy in func_11.



```
User
specify dates, options
SC_1
func_11(tour dates, service option)
  {
   //prepare phase
   func_21(dates, option, deadline, prepare)
   func_31(dates, option, deadline, prepare)
   func_41(dates, option, deadline, prepare)
   //change behaviour based on the results
   IF(all three functions return true) THEN {
    func_21(accept)
    func_31(accept)
    func_41(accept)
     IF(all three functions return true)
     Return reservation content; //success }
   ELSE    THEN {
    func_21(abort)
    func_31(abort)
    func_41(abort)
    Return failure reasons; // failure }
 }
```

```
SC_2
func_21(dates, option, deadline, prepare/commit)
  {
   IF(prepare){
   func_51(dates, deadline, prepare)
   func_61(dates, deadline, prepare)
   //Here decides which airline is selected
   IF (airline_A is chosen)    fun_61(abort)
   ELSE    func_51(abort)
   Return    reservation content; // to func_11 }

   IF(commit:accept){
   IF (airline_A is chosen)    func_51(accept)
   ELSE    func_61(accept) }

   IF(commit:abort){
   IF (airline_A is chosen)    func_51(abort)
   ELSE    func_61(abort) }
  }
```

Fig. 5. Pseudo code of func_11 and func_12.

After receiving the results from func_21, func_31, and func_41, func_11 can judge whether each reservation has been successfully made. If all reservations are successful, func_11 sends func_21(accept), func_31(accept), and func_41(accept) in the commit phase for unlocking all the locked reservation data after the prepare phase. If it is confirmed from the return value of each function that the accept requests for all the functions have been executed normally, func_11 notifies the user that the reservation has been successfully made and provides the reservation details. Conversely, if a process does not end normally in one of the functions one level down, these three functions are called with an abort request in the commit phase, and all the reservation processes that had been made are rolled back; func_11 then replies to the user with the reason why the reservation could not be made.

When func_21 is called with the prepare flag, func_51 and func_61, which are one level below func_21, are also called with the prepare flag. Then, func_21 receives the availability and conditions of each airline's reservation from each of those functions, and func_21 compares the two airlines based on the options/conditions the user wants. As a result, one of the airlines will be selected, so the company that is not selected will receive an abort request in the commit phase and cancel the reservation. Func_21 then returns the selected airline reservation to func_11. Although it is omitted from the code in Fig. 5, if neither company can offer a reservation, func_11 will notify the user of the reservation failure and the reason.

After that, if func_21 is called from func_11 with an accept flag in the commit phase, it issues an accept request in the commit phase to complete the booking to the function of the airline that made the reservation. Also, if func_21 receives an abort request, an abort request is also sent to the function of the booked airline to roll back the reservation. In this way, the whole cross-shard transaction process is controlled by the hierarchically top function called by the user; thus, the user does not need to specify the relationship among the shards involved in the transaction.

Furthermore, the hierarchical relationship among the functions involved in one cross-shard transaction can be determined by the programmers who write the cross-shard transaction. For instance, various other cross-shard transactions can be created by utilizing different combinations of the functions in the shards shown in Fig. 4. Therefore, a new cross-shard transaction can be created by the programmer determining the hierarchically top function and writing the code of the behavior of the top function including the calls of the underlying smart contract functions.

Besides utilizing the existing functions, the underlying smart contract functions can be added by other programmers who code the behaviors in the prepare phase and commit phase in individual functions. In the commit phase, the behaviors for the acceptance and abortion are separately written by the programmers. Each function is independent from the other functions even in the same cross-shard transaction, as long as they obey the determined input/output dataset of the functions; thus, programming flexibility and reusability of the functions are large.

C. Superiority of the proposed sharding for cross-shard transactions to other shardings

One way that the proposed sharding is superior to previously proposed shardings is that it has a much reduced risk of an eclipse attack. In it, only a node within the same shard to which the target validator belongs has a chance to become a neighbor node of the target validator. Therefore, if there are S shards and each shard has a similar number of validators, the probability that an eclipse attacker can become a neighbor of the target validator is approximately $1/S$ compared with the other shardings.

As well, the proposed sharding shuffles validators among multiple shards during the epoch interval; thus, if an attacking node succeeds in becoming one of the neighbors of the target validator, it will be deprived of its neighborship once it is shuffled out of the target validator's shard. By comparison, the previous shardings have no rule to replace neighbors based on shard membership, so the attacking node can remain a neighbor of the target for a much longer time.

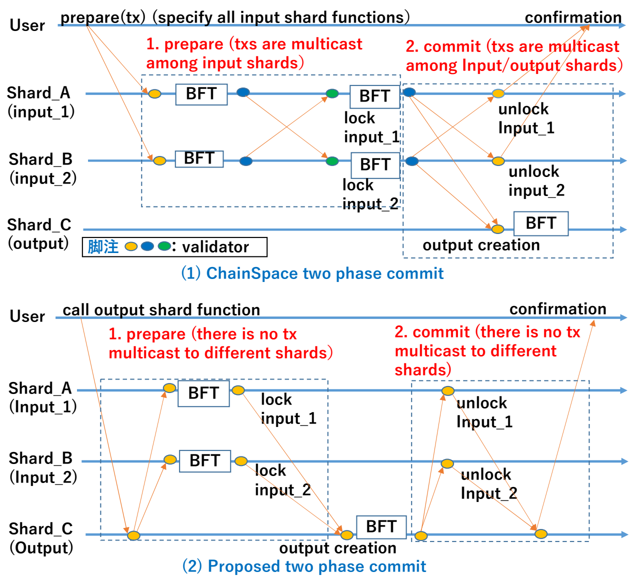

User
prepare(tx) (specify all input shard functions)
confirmation
1. prepare (txs are multicast among input shards)
2. commit (txs are multicast among Input/output shards)
Shard_A (input_1)
BFT
BFT
lock input_1
unlock Input_1
Shard_B (input_2)
BFT
BFT
lock input_2
unlock input_2
Shard_C (output)
BFT
output creation
脚注 ●●●: validator
(1) ChainSpace two phase commit
User
call output shard function
confirmation
1. prepare (there is no tx multicast to different shards)
2. commit (there is no tx multicast to different shards)
Shard_A (Input_1)
BFT
lock input_1
unlock Input_1
Shard_B (Input_2)
BFT
lock input_2
unlock Input_2
Shard_C (Output)
BFT
output creation
(2) Proposed two phase commit

Fig. 6. Comparison of two shardings dealing with a cross-shard transaction.

Another advantage that the proposed sharding has over its predecessors is in the two-phase commit. Here, shardings such as ChainSpace [6] and OmniLedger [7] have a two-phase commit whereby they deal with cross-shard transactions by having the user in the cross-shard transaction specify the shards involved, and the validators in these shards have to communicate with other shards to confirm transaction results in other shards by using the gossip protocol through its neighbors.

This subsection compares ChainSpace with the proposed sharding, because either can be applied to the account/balance blockchain model. As shown in Fig. 6(1), in ChainSpace, a user requests a cross-shard transaction (tx) by calling all the functions in the individual shards to create the inputs of the transaction. In this example, Input_1 is created in Shard_A by a validator called by the user and Input_2 is created in Shard_B by a validator called by the user, and the validators in each shard verify each of these inputs by using a consensus algorithm such as BFT (Byzantine fault-tolerant) through the gossip protocol.

After that, the verified inputs are multicast to validators in other input shards involved in the cross-shard transaction by using the gossip protocol, so that they will be verified in other related shards. At the same time, all the transaction results verified in the input shards are locked and not allowed to be accessed by other transactions until the transaction result is sent back to the user. After all the inputs have been verified by the validators in the input shards, they are multicast to the validators in the output shard, Shard_C, after which the output is created in Shard_C and sent back to the user.

Fig. 6(2) illustrates the cross-shard transaction management in the proposed sharding framework. Here, the user calls the smart contract function of Shard_C that accepts the service request with some options, such as the user's preferences and conditions, and the function returns the transaction result to the user. This hierarchically top function is conducted by one of the validators in Shard_C and it manages the two-phase commit of the transaction and calls the two one-level lower functions owned by the validators in Shard_A and the validators in Shard_B in the prepare phase. Each validator in Shard_A or in Shard_B receives the transaction request via the transaction pool for each shard, so it is called in a unicast manner and the gossip protocol is not used.

Each called function located in Shard_A or Shard_B creates input_1 and input_2, respectively; input_1 is verified by the validators in Shard_A and input_2 is verified by those in Shard_B through the gossip protocol in each shard. Then, input_1 and input_2 are locked and they are not allowed to be accessed by the other transactions until they are released in the commit phase. The validators in each shard, however, have no need to multicast the verified input to validators in other shards, because the function in Shard_C can receive the input results from the functions in Shard_A and Shard_B in the prepare phase, and it can judge if they have been successfully conducted.

The hierarchically top function located in Shard C can create the output of the transaction and returns the result to the user after unlocking the input data in the underlying shards, so there is no need to multicast the inputs or the output among validators in different shards. In this way, the proposed sharding avoids generating multicast traffic among different shards, and thus it dramatically lightens the data-transmission burden for each validator compared with ChainSpace.

In addition, from the user's perspective, it is difficult to recognize all the relationships among different smart contract functions which create transaction inputs and outputs. Moreover, as shown in the previous subsection, a hierarchical cross-shard transaction, whose hierarchical height is three or more, cannot be treated in the ChainSpace framework. On the other hand, in the proposed sharding, each user specifies only the hierarchically top function and its service parameters as the function arguments. Thus, the relationship of the functions involved in the cross-shard transaction and its data atomicity can be managed hierarchically by the involved functions according to the user's requirements.

V. Evaluations

The main feature of the proposed sharding is that each validator keeps all of its neighbors within its shard, so it can choose any neighbor selection method to determine the neighbors for each validator within a shard. However, because of the frequent changes of the neighbors for each validator in the proposed sharding, a faster neighbor selection is preferable. Here, the random neighbor selection method, which is used in the default Bitcoin node setting [16], is applicable because it also has each validator select its neighbors swiftly as in the region-based method. Thus, the evaluations compared simulated networks using the random neighbor selection method and region-based neighbor selection method with and without the proposed sharding and examined the effect of the proposed sharding on block propagation and queueing delays.

A. Simulation environment

We used real-time data collected from the Ethereum main network [26] in Oct. 2023 to make the simulation network. At the time, 4,228 validators were participating in the network, and the validator rates for the individual regions in the world were as follows: 43.61% in North America (NA), 41.86% in Europe (EU), 4.92% in East Asia (EA), 4.42% in South East Asia (SEA), 3.34% in Australasia (AUS), 0.78% in South America (SA), 0.76% in the Middle East (ME), and 0.31% in Africa (AF). In the simulation, 4,228 nodes were probabilistically allocated to the above regions in the proportions listed above.

Whenever a network is randomly created in the manner mentioned above, the created nodes are randomly allocated to multiple shards, where a shard-member upper limit M is set for each shard. In the simulation, we set this shard member limit to $M = \lceil 4228/S \rceil$, where $\lceil\,\rceil$ denotes the ceiling function and S is the number of shards. Once the nodes were allocated to individual shards, they became validators for the allocated shard. After that, each one chose its neighbors using one of the four neighbor-selection methods, namely random selection with and without the proposed sharding and the region-based selection with and without the proposed sharding.

To compare the four methods fairly, the upper limit of outbound/inbound neighbors for each validator was set to 8. In addition, in the simulation of the region-based selection method without the proposed sharding, at most two of the inbound and outbound neighbors were inside neighbors. On the other hand, in the simulation of the region-based selection with the proposed sharding, each validator had at most one inside neighbor when $M<1000$; otherwise each validator had at most two inside neighbors. Note that these maximum numbers of the inside nodes were determined in accordance with the recommendation in [21].

We set the average propagation delays between pairs of Ethereum main-network nodes in or among the eight regions on the basis of ping transmission time data measured by the site [27]. In particular, we gave each link in the simulations a propagation delay in the range of 90–110% of the corresponding value for the region pair in the delay matrix shown in Table I.

TABLE I. AVERAGE PROPAGATION DELAY MATRIX AMONG REGIONS IN ETHEREUM NW IN 2023 (UNIT: MS)

	NA	EU	EA	SEA	AUS	SA	ME	AF
North America (NA)	10.12	43.13	91.93	136.81	123.2	74.41	82.7	86.49
Europe (EU)	43.13	6.52	133.17	81.44	151.27	114.05	47.15	45.07
East Asia (EA)	91.93	133.17	16.76	56.58	95.85	176.58	157.45	172.27
South East Asia (SEA)	136.81	81.44	56.58	40.83	94.15	184.09	85.5	133.68
Australasia (AUS)	123.2	151.27	95.85	94.15	24.51	187.91	179.34	194.21
South America (SA)	74.41	114.05	176.58	184.09	187.91	15.03	148	146.14
Middle East (ME)	82.7	47.15	157.45	85.5	179.34	148	86.17	86.59
Africa (AF)	86.49	45.07	172.27	133.68	194.21	146.14	86.59	87.32

Besides propagation delays, queuing delays should be also considered in a blockchain network. Meng et al. showed that even for a small blockchain network with less than five validators, the validation time for a block is lengthened by more than 10 *ms* by queueing delays if its block size is increased [28]. Therefore, to each validator, we applied the M/M/1 queuing model [29], where V_{q_delay} and $V_{q\_delay\_with\_proposed}$ are the average queuing delay for each validator without and with the proposed sharding:

$$V_{q\_delay} = \frac{B_{size} * S}{Q_{cap} * T}, \qquad (1)$$

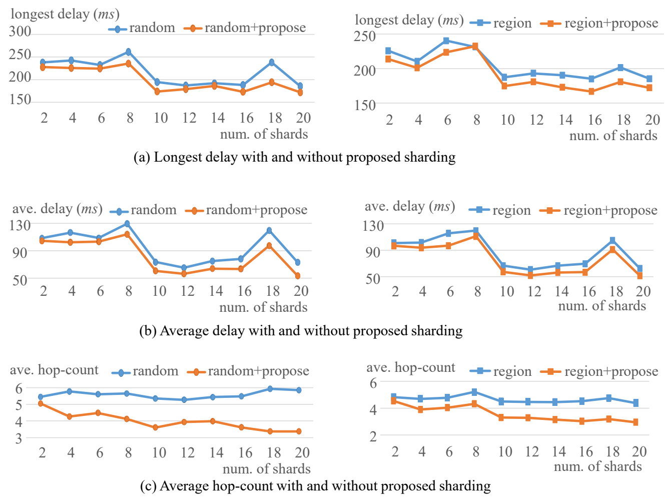
$$V_{q\_delay\_with\_propose} = \frac{B_{size}}{Q_{cap} * T}. \qquad (2)$$

Here, B_{size} indicates the block size, Q_{cap} is the queue capacity for each validator, which we set to $Q_{cap} = 100MB/s$, and T is the block creation interval, which is 12 s in the current Ethereum network. The difference between (1) and (2) comes from the fact that, in the proposed sharding, each validator does not have to forward the blocks for the other shards, while in the other shardings, each validator has to forward the blocks for all the shards; thus, the block traffic is multiplied by S, which is the number of shards.

After assigning the propagation and queuing delays to the corresponding links and validators, one source validator was randomly chosen from the 4,228 nodes. Then, after all the neighbors were set for all the validators by using each selection method, the smallest delay routes from the source validator to all the other validators in the same shard were calculated and they were determined as the delays of the block transmission from the source validator to the others.

B. Evaluation results and their analyses

Figure 7 compares the longest/average block delays and the average hop counts of the four neighbor-selection methods. Here, B_{size} was set to 0.02B, the current Ethereum block size. For each point on the horizontal axis, which is the number of shards (2–20), the 4228 validators were allocated to the eight regions with the probabilities mentioned above; then, the source validator was randomly selected and these values are measured. We conducted this data sampling two times for each point on the horizontal axis (the graphs hence plot the averages of the two delays).



(a) Longest delay with and without proposed sharding

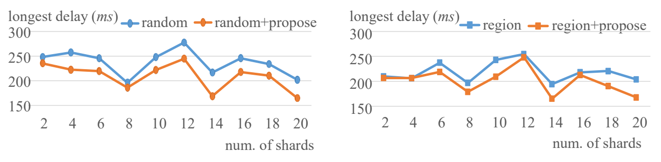
(b) Average delay with and without proposed sharding

(c) Average hop-count with and without proposed sharding

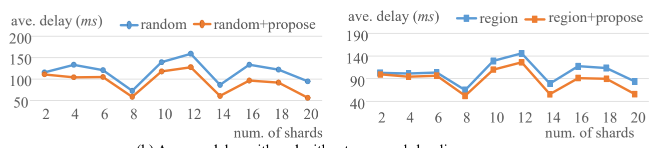
Fig. 7. Simulated delays and hop counts for $B_{size} = 0.02MB$.

As shown in the figure, the longest and average delays varied disproportionately with the number of shards, mainly because of the random selection of the source node. The proposed sharding, however, reduced the longest delays in both selection methods by 7.6% compared with not using it, and it reduced the average delays of the random (region-based) method by 14.1% (12.9%). It reduced the delays because the nodes did not have any unnecessary neighbors in other shards and thus there were faster routes on which to send a block to each destination.

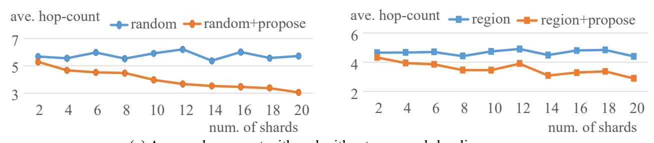
The average hop counts from the source to the other validators are shown in (c); this graph demonstrates that the hop-count gap between the methods with and without the proposed sharding became wider as the number of shards increased. This is because each validator in the proposed sharding had all of its neighbors in its own shard.



(a) Longest delay with and without proposed sharding



(b) Average delay with and without proposed sharding



(c) Average hop-count with and without proposed sharding

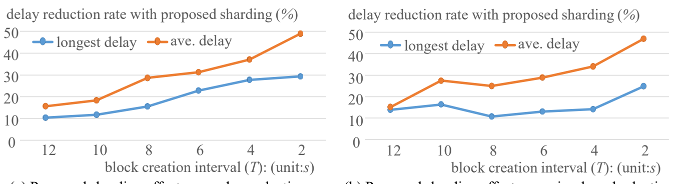
Fig. 8. Simulated delays and hop counts for $B_{size} = 1MB$.

Besides sharding, the block size B_{size} is another important influence on the TPS (transactions per second) of the Ethereum network. Thus, we also evaluated the delays and hop counts in the case of a different block size, $B_{size} = 1MB$. The results are shown in Fig. 8, where the other conditions are the same as in Fig. 7 except for the block size. The effectiveness of the proposed sharding becomes clearer: it reduced the longest delays (a) by 11.9% for the random method and 8.3% for the region-based method, and it reduced the average delays (b) by 21.6% for the random method and 17.2% for the region-based method. Its effect was especially large when there were many shards. For example, when the shard number was 20, it shortened the delay of (b) by 40.4% for the random method and 33.6% for the region-based method.

The results in Figs. 7(c) and 8(c) show that a larger number of shards is beneficial for the proposed sharding; here, the average hop-count gap between cases with and without the proposed sharding became larger as the number of shards was increased. If the hop count is large between a pair of validators, the queueing delay between them becomes longer especially for larger block sizes.

Another typical way to enhance the TPS for a blockchain is shortening the block creation interval, which is T in formula (1) and (2). Thus, we conducted a simulation in which we reduced the value of T from 12 s to 2 s in decrements of 2 s. Fig. 9 shows the percentages by which the proposed sharding reduced the longest and average delays of block transmissions among the validators in the simulation environment. The number of shards was set as S=10, and the block size was set as $B_{size} = 1MB$.

From (1) and (2), it is clear that a smaller T increases the queuing delay difference between with and without the proposed sharding. The evaluation results in Fig. 9 by and large follow these formulas for the most part, though in (b) some plots have smaller reduction rates (e.g. T=8 has a smaller reduction rate than T=10) because of the random source validator selection for each plot.



(a) Proposed sharding effect on random selection (b) Proposed sharding effect on region-based selection

Fig. 9. Delay reduction rates with proposed sharding versus block creation interval for $B_{size} = 1MB$.

The results described in this section indicate that the proposed sharding enables Ethereum blocks to be transmitted to the other validators in the same shard much faster than the other shardings do with any block size and block creation interval. However, its effect grows as the block size becomes larger and the block creation interval becomes shorter.

VI. Conclusions

This paper proposed a new sharding architecture in which smart contract functions in different shards cooperate with each other hierarchically to accomplish the two-phase commit for a cross-shard transaction. The proposed sharding removes the need for multicasting validation results in one shard to the other involved shards; thus, the burden on individual validators is significantly reduced. In addition, the hierarchically top function receiving a user request manages the cross-shard transaction without asking the user to specify all of the involved shards and functions. Therefore, the user's burden is also reduced.

The proposed sharding also enables each validator to confine its neighbors within its own shard, which is effective for reducing the risk of the Eclipse attacks and enhancing the transmission speed of transactions/blocks in each shard. In order to verify this effect, we applied the proposed sharding to the random and region-based neighbor-selection methods in computer simulations. The results of the evaluation demonstrated its superiority to other shardings in terms of shorter block transmission delays and smaller hop-counts within a shard. In addition, it was also turned out that the proposed sharding's effect on reducing the delays is enhanced if the block size is increased or the block creation interval is shortened.

In the future, we will analyze the theoretical end-to-end transaction/block delays between the source and other validators when the proposed sharding is applied while varying the network conditions.

References

[1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” Tech. Rep., 2008. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[2] S. Gherghelas, “State of the Dapp Industory in Q2 2023,” July 2023. [Online]. Available: https://dappradar.com/blog/state-of-the-dapp-industry-in-q2-2023

[3] D. Reijsbergen, S. Sridhar, B. Monnot, S. Leonardos, S. Skoulakis, and G. Piliouras, “Transaction fees on a honeymoon: Ethereum’s EIP-1559 one month later,” *IEEE International Conference on Blockchain*, 2021.

[4] S. Wels, “Guaranteed-tx: The exploration of a guaranteed cross-shard transaction execution protocol for Ethereum 2.0,” Master’s thesis, University of Twente, 2019.

[5] D. Tennakoon and V. Gramoli, “Dynamic Blockchain Sharding,” *5th International Symposium on Foundations and Applications of Blockchain (FAB)*, no. 1, pp. 1–17, 2022.

[6] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, “Chainspace: A sharded smart contracts platform,” *arXiv preprint*, arXiv:2203.12323, 2022.

[7] E. K.-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, “Omniledger: A secure, scale-out, decentralized ledger via sharding,” *IEEE Symposium on Security and Privacy,* pp. 583–598, 2018.

[8] M. Tran, I. Choi, G. J. Moon, A. V. Vu, and M. S. Kang, “A stealthier partitioning attack against Bitcoin peer-to-peer network,” *IEEE Symposium on Security and Privacy*, 2020.

[9] A. K. Yildiz, A. Atmaca, A. O. Solak, Y. C. Tursun, and S. Bahtiyar, “A trust based DNS system to prevent Eclipse attack on blockchain networks,” *15th International Conference on Security of Information and Networks*, Nov. 2022.

[10] M. Zamani, M. Movahedi, and M. Raykova, “RapidChain: Scaling blockchain via full sharding,” *ACM SIGSAC Conference on Computer and Communications Security*, pp. 931–948, Oct., 2018.

[11] S. Kim, J. Song, S. Woo, Y. Kim, and S. Park, “Gas consumption-aware dynamic load balancing in Ethereum sharding environments,” *IEEE International Workshops on Fundations and Applications of Self Systems*, 2019.

[12] N. Okanami, R. Nakamura, and T. Nishide, “Load balancing for sharded blockchain,” *International Conference on Financial Cryptograpy and Data Security*, pp. 512–524, 2020.

[13] Z. Hong, S. Guo, E. Zuou, W. Chen, H. Huang, A. Zomaya, “GriDB:Scaling blockchain database via sharding and off-chain cross-shard mechanism,” *Proc. of the VLDB Endowment*, vol. 16, no. 7, pp. 1685–1698, March, 2023.

[14] Y. Shahsavari, K. Zhang, and C. Talhi, “A theoretical model for folk analysis in the Bitcoin network,” *IEEE Blockchain*, 2019.

[15] A. Kertesz and H. Baniata. “Consistency Analysis of Distributed Ledgers in Fog-enhanced Blockchains,” *International European Conference on Parallel and Distributed Computing (Euro-Par 2021)*, vol. 27, 2021.

[16] Bitcoin/bitcoin: https://github.com/bitcoin/bitcoin, Aug. 2023.

[17] Y. Aoki and K. Shudo, “Proximity neighbor selection in blockchain network,” *IEEE International Conference on Blockchain*, 2019.

[18] K. Wang and H. S Kim, “FastChain: Scaling blockchain system with informed neighbor selection,” *IEEE International Conference on Blockchain*, 2019.

[19] S. Park, S. Im, Y. Seol, and J. Paek, “Nodes in the Bitcoin networks: Comparative measurement study and survey”, *IEEE Access*, vol. 7, pp. 57009–57022, April 2019.

[20] H. Baniata, A. Anaqreh, and A. Kertesz, “Dons: Dynamic optimized neighbour selection for smart blockchain networks,” *ELSEVIER, Future Generation Computer Systems*, vol. 130, pp. 75–90, 2022.

[21] H. Matsuura, Y. Goto, and H. Sao, “New neighbor selection method for blockchain network with multiple regions,” *IEEE Access*, vol. 10, pp. 105278–105291, Sep. 2022.

[22] A.-M. Kermarrec and M. v. Steen, “Gossiping in distributed systems,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 2–7, Oct. 2007.

[23] Alchemy Team, “Ethereum sharding: An introduction to blockchain sharding,” May, 2022. [Online]. Available: https://www.web3.university/article/ethereum-sharding-an-introduction-to-blockchain-sharding

[24] V. King and J. Saia, “Breaking the $O(n^2)$ bit barrier: Scalable byzantine agreement with an adaptive adversary,” In *Proc. Of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 420–429, 2010.

[25] D. Tennakoon, Y. Hua, and V. Gramoli, “Collachain: A BFT collaborative middleware for decentralized applications,” *arXiv preprint*, arXiv:1708.03778, 2017.

[26] Ethernodes org, “Ethereum Mainnet Statistics,” https://www.ethernodes.org/, Oct. 2023.

[27] Global Ping Statistics: https://wondernetwork.com/pings, July 2022.

[28] R. A. Memom, J. P. Li, and J. Ahmed, “Simulation model for Blockchain systems uing queuing theory,” MDPI Electronics, vol. 8, no. 2, Aug. 2019. [Online]. Available: https://www.mdpi.com/2079-9292/8/2/234

[29] T. Meng, Y. Zhao, K. Wolter, and C.-Z. Xu, “On consortium blockchain consistency: A queueing network model approach,” IEEE Trans. On Parallel and Distributed Systems, vol. 32, no. 6, pp. 1369–1382, June 2021.