Article | Received 30 December 2023; Accepted 27 March 2024; Published 12 April 2024
https://doi.org/10.55092/blockchain20240002

Frustum: achieving high throughput in blockchain systems through hierarchical and pipelined sharding

Yukun Xu[1,†], Wenhan Wu[1,†], Yili Gong[1,*], Kanye Ye Wang[2], Chuang Hu[1] and Dazhao Cheng[1]

[1]School of Computer Science, Wuhan University, Wuhan, China
[2]Faculty of Science and Technology, University of Macau, Macao, China
[†]Co-first Author

* Correspondence author; E-mail: yiligong@whu.edu.cn.

Abstract: Sharding, breaking nodes into smaller groups, aims to enhance the scalability of traditional blockchain systems by allowing parallel transaction processing. However, existing sharding methods face challenges, including heavy inter-shard communication, re-sharding overhead, and low consensus concurrency. These limitations ultimately result in less desired system performance. To address these challenges, we propose Frustum, a novel hierarchical and pipelined sharding blockchain system. It separates shards into two layers: top L-Shard and base F-Shards. In each round, a global leader is elected from L-Shard and broadcasts a new block to F-Shard nodes, negating the need for final committee confirmation and simplifying the consensus process. Additionally, Frustum adopts a random re-sharding mechanism to mitigate the re-sharding overhead issue. Finally, Frustum employs a pipelined structure for enhanced consensus concurrency. Our Frustum prototype demonstrates a substantial performance boost, improving transaction throughput by 2.79 and 1.68 times over existing sharding systems with 16 shards and 1024 nodes.

Keywords: blockchain sharding; complete pipeline; Frustum

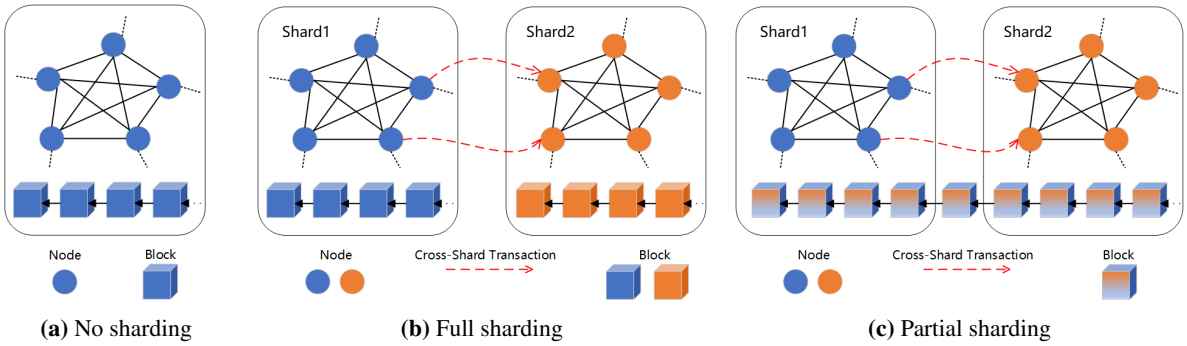
1. Introduction

Blockchain technology has gained widespread adoption in a variety of fields such as digital cryptocurrency [1], information security, and the Internet of Things [2] due to its inherent features of decentralization, data transparency, and security [3]. However, in order to achieve a high level of security in a large-scale decentralized environment, blockchain technology often suffers from poor transaction throughput and latency [4]. This is because blockchain consensus protocols require all nodes to verify and store all transactions, and every consensus message must be broadcast across the entire blockchain network (see Figure 1a). For instance, Bitcoin [1], one of the most popular blockchain system, takes an average of 10 minutes to produce a block [5], which is far from sufficient for a practical financial system.

Sharding is considered a promising scheme to improve blockchain systems' performance in terms of transaction throughput and latency [6]. It partitions the blockchain network into smaller *shards*. Each shard hosts a subset of the overall network, independently processes transactions and maintains its own copy of the blockchain ledger. The key benefit of sharding lies in its ability to improve the scalability and performance of the blockchain network by reducing the computational and storage demands associated with transaction validation [7].

Full sharding and *Partial sharding* are two state-of-the-art approaches for blockchain sharding. Full sharding (see Figure 1b), like OmniLedger [8] and RapidChain [9], divides

the nodes into multiple isolated shards, each with a distinct blockchain for the transactions that it processes. The nodes within a shard are responsible for the consensus of transactions submitted to this shard, including verification, storage and communication. Intrinsically, full sharding reduces the number of nodes involved in a transaction, thereby improving the intra-shard transaction performance. However, it needs to process several sub-transactions per cross-shard transaction, since it divides original cross-shard transactions into multiple sub-transactions, which seriously degrades the cross-shard performance in terms of throughput and confirmation latency. To this end, partial sharding (see Figure 1c) is proposed to mitigate the cross-shard transaction issue. It distributes the transactions to different shards and each shard maintains a globally consistent blockchain so that there are no cross-shard transactions in the system, such as Elastico [10] and Zilliqa [11].



(a) No sharding **(b)** Full sharding **(c)** Partial sharding

Figure 1. Illustration for different sharding blockchain systems.

Existing partial sharding suffers from burdensome inter-shard communication, heavy re-sharding overhead and low consensus concurrency. First, a block produced by a shard should be synchronized with all nodes in other shards to maintain a single chain. The inter-shard communication increases with the number of nodes and transactions. We find that the communication latency may become the bottleneck. Second, partial sharding also requires frequent re-sharding to improve the system's ability to resist Byzantine attacks [12]. Nodes in a blockchain system may act maliciously and try to undermine the security and correctness of the network, leading to system failure or data tampering. If the number of malicious nodes in a shard exceeds 1/3, the attack will be successful. Therefore, frequent re-sharding can prevent the attacker from gaining knowledge of a shard's composition, thus resisting attacks. However, re-sharding requires complex node verification through solving the PoW problem [13], which introduces significant computing latency. Our measurement study on Elasticso [10] shows that the re-sharding latency can reach up to 94.03% of the total epoch time (see Section 2.3. for details). Third, in each epoch, shard formation, shard overlay setup, intra-shard consensus, final consensus broadcast, and epoch randomness generation proceed sequentially. The low concurrency of the consensus process severely hinders the throughput performance, leaving an opportunity for parallelism.

Aiming at the above issues in partial sharding, we propose *Frustum*, a novel hierarchical and pipelined sharding blockchain system. We reduce inter-shard communication by designing a hierarchical sharding architecture whose shape resembles a frustum. The bottom circular base consists of *Foundation shards* (F-Shard). In each F-Shard, a node is selected as *leader* to coordinate intra-shard consensus and block committing. The top circular base is the *leader shard* (L-Shard), consisting of all F-shard leaders. Inter-shard communication is confined among L-Shard nodes, which significantly reduces the number of participating nodes and alleviates communication pressure.

We propose a random re-sharding mechanism to mitigate the overhead of frequent re-sharding. While re-sharding every epoch lowers the attacker's probability of detecting the shard composition, randomly selecting timings for re-sharding increases the indeterminacy

and achieves the similar effect with low overhead.

In our research, we have developed a complete pipelined consensus protocol with the objective of increasing consensus concurrency. In contrast to the conventional sequential consensus approaches, our protocol employs a pipelining strategy, which allows for the concurrent execution of different stages from sequential epochs.

Through a systematic investigation, proper consensus stages for pipelining are identified. Accordingly, we have introduced a fine-grained pipelining transaction processing protocol, which significantly improves throughput and efficiency in blockchain systems. Our research findings and the proposed protocol play a significant role in advancing the field of blockchain technology, addressing vital concerns related to performance and scalability.

We also design a complete pipelined consensus protocol to increase concurrency. Unlike existing sequential blockchain consensus algorithms, Frustum orchestrates the consensus process in a pipelining way, which allows for overlapping of the different stages of sequential epochs. We investigate which stages can be executed concurrently, and proposed a fine-granularity pipelining transaction processing protocol.

Finally, based on PBFT [12], a standard byzantine [14] agreement protocol, we implement a prototype of Frustum and evaluate its performance. Experiments show that Frustum can process more than 4600 tx/sec in a network with 16 shards, which improves the transaction throughput by 2.79 $\times$ and 1.68 $\times$ compared to the state-of-the-art sharding systems.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 provides the details of the Frustum design. Section 4 gives the system analysis for Frustum. Section 5 evaluates the performance of Frustum compared with other sharding blockchain systems by experiments. Section 6 presents the related work. Finally, Section 7 concludes this article.

2. Background and motivation

This section first introduces the concepts of blockchain systems, which have attracted extensive attention from the industry and academia. Then we present a measurement study and workflow analysis of a typical state-of-the-art partial sharding blockchain system Elastico [10].

2.1. Background on blockchain systems

Blockchain is a distributed storage technology in which data are stored in the form of blocks [15]. Each block is divided into a block header and a block body. A block header records the birth date of the block, the hash digest of the previous block, the hash digest of this block, and other ancillary information [16]. A block body holds the specific transactions which consist of digital currency transactions between different accounts that are anonymous strings of characters in Bitcoin [17].

Each node participating in the blockchain system keeps an identical copy of the blockchain data and always synchronizes the new blocks generated by miners to ensure the consistency of data. In traditional blockchain systems, particularly those utilizing Proof of Work (PoW), miners are required to consume a significant amount of computational power in order to compete for the privilege of adding a new block to the chain [18]. However, modern consensus mechanisms, such as Proof of Stake (PoS), Delegated Proof of Stake (DPoS), and others, provide alternative approaches that do not necessitate this intensive utilization of computational resources [19, 20]. In PoS-based systems, for example, the authority to validate transactions and generate new blocks is determined by one's stake in the network, resulting in significantly reduced energy consumption and expedited transaction validation [19]. Miners who mine a new block will get a block reward. Miners who lose the competition will continue to mine on top of the new block, thus continuing to form newer blocks and the blockchain keeps growing [21]. Processing a block can be divided into four steps, as shown in Figure 2.

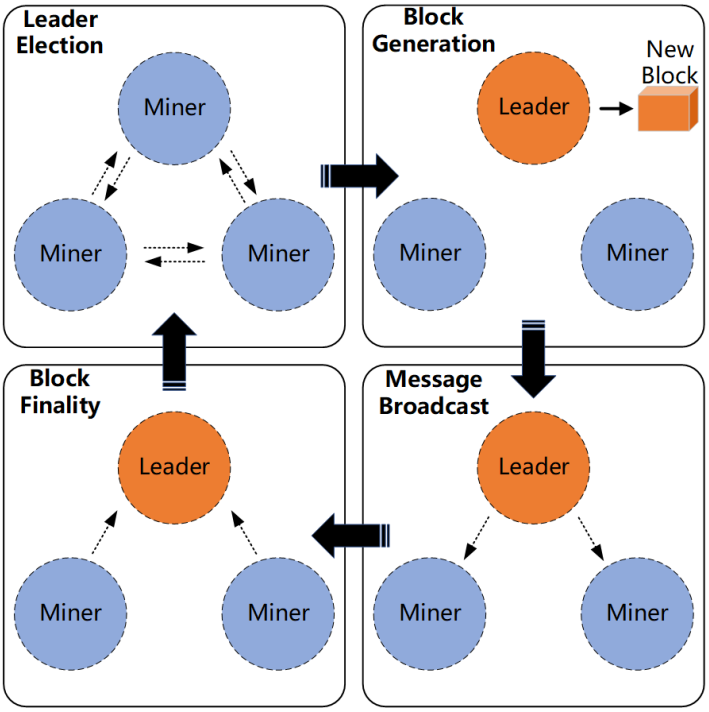


Figure 2. Four-stage processing of general blockchain consensus.

Leader Election: Each block is mined by a miner, and the blockchain system needs a method of filtering miners to elect a unique leader to take on the job of generating new blocks. In consensus mechanisms such as Practical Byzantine Fault Tolerance (PBFT) [22] and Delegated Proof of Stake (DPoS), the leader election is an explicit phase where nodes select a leader through a designated process. In Proof of Stake (PoS) mechanisms, the election is often influenced by factors such as the stake's size and duration held by the nodes. Conversely, in Proof of Work (PoW), the leader emerges implicitly by solving cryptographic puzzles, with any miner capable of becoming a leader upon successfully completing such a puzzle.

Block Generation: The winning node in the previous Leader Election phase, called the leader, takes on the job of packaging transactions to generate a new block. The leader maintains a transaction pool, which holds transactions that have been verified as legitimate, waiting for being packed into a new block according to certain rules. These transactions are sent directly to the leader by clients (also known as wallets [23]) or routed from other nodes. This phase, which involves the elected leader packaging transactions to create a new block, is common in PoS, DPoS and PBFT mechanisms. Particularly, block generation in PoW is tightly coupled with leader election, occurring simultaneously as the computational puzzle is solved.

Message Broadcast: After generated, a new block needs to be confirmed by a majority of the consensus members. Therefore, the leader broadcasts the new block to the other consensus members in the network. In PBFT, PoS, and DPoS, this involves a series of protocol communications to reach an agreement on the block. In PoW, the process is relatively straightforward as the new block is disseminated through the network and validated.

Block Finality: When the leader and most other nodes reach consensus on the new block, they add the new block to the blockchain, which means that the transactions in the new block are actually submitted in success. In PBFT, PoS, and DPoS, when a majority of nodes reach a consensus on the new block, it is appended to the blockchain, signifying the ultimate commitment of the transactions within. In PoW, finality is typically achieved when the block receives a sufficient number of confirmations through successive block additions to the chain.

Blockchain systems generally cycle through the above four steps, with new blocks con-

stantly being submitted and blockchain data growing.

2.2. A measurement study on Elastico

Elastico [10] is a partial blockchain sharding technique that aims to provide high scalability and low latency for blockchains. In this section, we conduct a performance measurement study on Elastico to investigate the impact of inter-shard communication and re-sharding.

Measurement setup. We implement Elastico system and calculate the inter-shard communication overhead and the ratio of time taken for re-shard in one epoch. For the transaction dataset, we use a real blockchain transaction trace, which is extracted from XBlock-ETH [24]. We randomly divide the original dataset into three parts, named Dataset 1, Dataset 2 and Dataset 3. We evaluate the Elastico system on a workstation with 20 CPU cores (@3.7GHz) and 125GB memory.

The impact of inter-shard communication. Table 1 shows the number of inter-shard messages per node and the bandwidth used per node. In different network sizes, the number of messages per node exceeds 1000, and the bandwidth consumed at each node fluctuates around 5 MB per node. Such communication overhead will seriously affect the system performance.

The impact of re-sharding. Figure 3 shows the Cumulative Distribution Function (CDF) of re-sharding latency over three datasets. We can observe that the re-sharding latency severely degrades the system performance in terms of transaction latency. For instance, only 22.62%, 23.15%, and 51.07% of the re-sharding latency ratios of Dataset 1, Dataset 2, and Dataset 3 are smaller than 20% of the total epoch time. More than 39.75%, 39.32%, and 17.77% of the re-sharding latency ratios of Dataset 1, Dataset 2 and Dataset 3 are greater than 50% of the total epoch time.

Table 1. Measurement results of inter-shard communication.

Network Size	Messages per Node	Bandwidth Consumption
100	2500	5.07 MB
200	1879	4.99 MB
400	1508	4.93 MB
800	1479	5.14 MB
1000	1463	5.10 MB

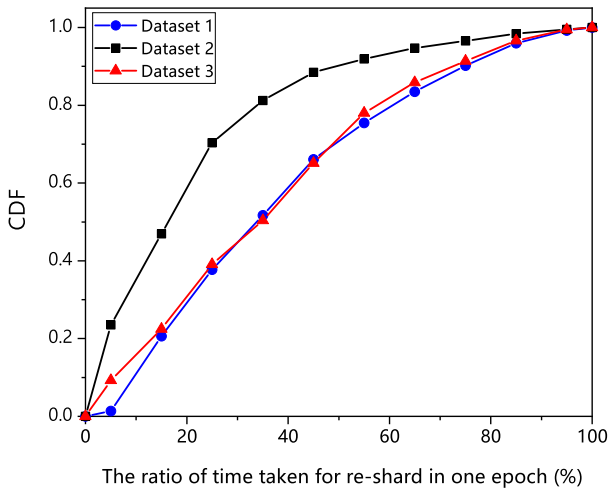


Figure 3. The CDF of re-sharding latency.

We can also see that re-sharding becomes the system bottleneck for part of epochs. For example, the maximum ratios between the re-sharding latency and the total epoch time are 93.47%, 94.03%, and 92.22% over three Dataset 1, Dataset 2, and Dataset 3, respectively.

2.3. An analysis study on Elastico

In this section, we analyze the consensus flow of Elastico, which motivates us to improve the concurrency of the partial sharding blockchain system.

As shown in Figure 4, the workflow of Elastico is as follows: 1) Identity Establishment and Committee Formation, each node chooses a local identity information group (IP, PK) which represents their IP address and public key [25] respectively. In order for the system to recognize their identity, each node needs to find a PoW solution that is related to the *epochRandomness* and node identity information. The *epochRandomness* will be generated in the last step of the previous epoch. After establishing their identity, the system divides nodes into 2^s committees by the last s bits of their node ID; 2) Overlay Setup for Committees, the first c nodes that establish identity and join the system form the directory committee. All nodes establish peer-to-peer connections with other nodes belonging to the same committee by contacting the directory committee; 3) Intra-Committees Consensus, transactions are assigned to a specific committee based on certain rules, and then the committee runs the PBFT consensus algorithm internally. If the transaction is validated, all nodes within the committee sign the transaction; 4) Final Consensus Broadcast, a final committee is randomly selected in the system, which then verifies that a transaction has been signed by more than half of the members of the corresponding committee. After successful validation, the transaction is packaged into a block and added to the blockchain; 5) Epoch Randomness Generation, the final committee will generate an *epochRandomness*, which will be used for identity establishment in the next epoch.

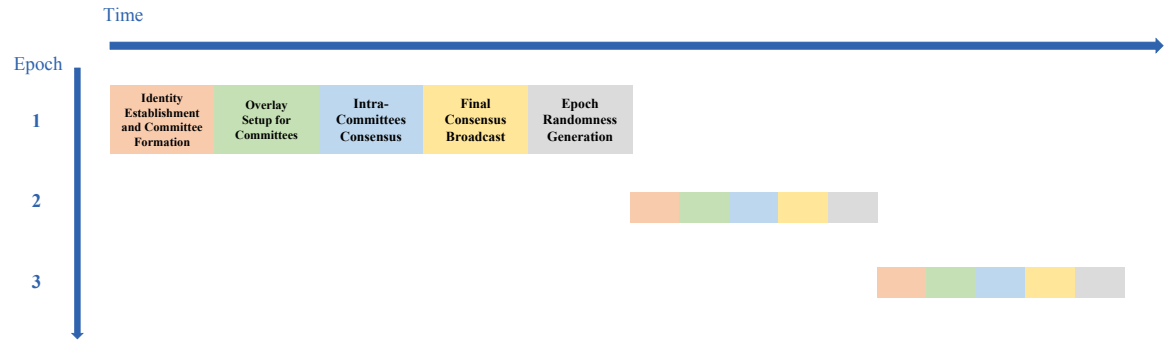


Figure 4. Consensus flow of Elastico.

Elastico performs the above steps in a serial manner within each epoch as well as between each epoch. Specifically, within each epoch, Elastico executes the five steps sequentially; only when the five steps of the current epoch are completed can the five steps of the next epoch be executed. However, this serial execution method has low parallelism, providing us with an opportunity to increase parallelism and improve the overall transaction throughput of the system. For instance, we can start the next round of Identity Establishment and Committee Formation right after finishing the same step of the current epoch.

3. Frustum design

This section makes assumptions about the security of the system and presents the system model of Frustum. Then the details of the Frustum random re-sharding mechanism and consensus protocol are elaborated, including the complete process of transaction handling

from submission to finalization into the blockchain.

3.1. Problem definition

Frustum system seeks to address the scalability and energy efficiency challenges prevalent in traditional blockchain consensus mechanisms like Proof of Work. Specifically, the system is designed to enable a higher transaction throughput without exponentially increasing the computational power required for consensus. Frustum is defined by its inputs, outputs, and the hierarchical structure of nodes and shards.

The system operates on a set of inputs comprising transaction requests, denoted as $T_{\text{req}} = \{t_1, t_2, ..., t_k\}$, where each t_i represents a digitally signed transaction request, and inter-node messages, represented as $M = \{m_1, m_2, ..., m_j\}$, which are used for transaction verification and consensus. The outputs of the system include a block of validated transactions, $B_{\text{val}} = \{t_{v1}, t_{v2}, ..., t_{vm}\}$, and an updated system state, represented as $Sys_{\text{new}} = Sys_{\text{old}} \oplus B_{\text{val}}$, where $\oplus$ denotes the state update operation.

We define the system's nodes and their organization into shards. The network is composed of a set of nodes, symbolized as $N = \{N_1, N_2, ..., N_n\}$, which are fundamental units responsible for processing transactions, maintaining ledger integrity, and executing consensus protocols. These nodes are organized into distinct shards, denoted as $S = \{S_1, S_2, ..., S_m\}$, with the intention of parallelizing transaction processing and enhancing system scalability. During a random round r reshard, the allocation of nodes to shards is determined by a hash function applied to the node's public key, $Alloc(N_i) = hash(pk_i) \mod 2^s$, where s represents the sharding parameter. This architecture allows the Frustum system to efficiently distribute workload across shards, thus addressing scalability while ensuring the security and decentralization of the blockchain.

In the Frustum blockchain system, the consensus process is defined streamlined yet robust, comprising several key steps to safeguard the network's integrity and consensus. It begins with the Global Leader Election, where a global leader, L_{global}, is chosen from the L-shard through the election function $elect(L)$. This leader is crucial for orchestrating the consensus across the network. Next, during the Block Generation phase, a candidate block, B_{gen}, is formed by compiling transaction requests, T_{req}, using the packing function $pack(T_{req})$. This block contains the transactions proposed for the blockchain. The Consensus Protocol follows, initiating with the Pre-prepare step where B_{gen} is shared with leaders via the distribution function $distribute(B_{gen}, L)$. During the Prepare phase, shard followers, F, authenticate B_{dist} using $verify(B_{dist}, F)$, ensuring only valid transactions proceed. The Commit phase depends on the outcome of verifications. If the count of nodes that verified the block, $|Verified|$, surpasses the supermajority threshold, $\lambda|F|$, B_{gen} is committed to the blockchain as B_{val}. If not, it's discarded. This system guarantees consensus is only achieved with ample node agreement. The Consensus Flow is formally encapsulated as:

$$Consensus(B_{gen}) = \begin{cases} Sys_{new} = Sys_{old} \oplus B_{val} & \text{if } |Verified| > \lambda|F| \\ \text{Discard } B_{gen} & \text{otherwise} \end{cases}$$

This represents a binary consensus decision: a block is either accepted and updates the state to $Sys_{new} = Sys_{old} \oplus B_{val}$ if it meets validation, or discarded. The parameters λ is crucial, denoting the supermajority threshold, the verification count, and the system's states pre and post-consensus. This framework ensures a secure, democratic, and efficient consensus route in the Frustum blockchain system.

3.2. Security assumptions and properties

The security guarantees established in Frustum rely on certain key assumptions to ensure the robustness of the system. We make three assumptions related to the limited presence of

malicious nodes, the reliability of digital signatures for authentication, and the effectiveness of the system's response to potential tampering. These assumptions collectively contribute to the establishment of a secure consensus process within the Frustum blockchain system.

- **Robust Digital Signature Authentication ($\mathscr{A}_1$).** Let $\mathscr{D}_s$ denote the digital signature mechanism. The assumption $\mathscr{A}_1$ posits that $\mathscr{D}_s$ functions correctly, enabling clients to authenticate requests ($\mathscr{R}_c$) and honest nodes to verify these requests' authenticity ($\mathscr{V}_r$). This assumption is pivotal, forming the cornerstone of Frustum's consensus security.
- **Limited Malicious Nodes ($\mathscr{A}_2$).** Denote by $\mathscr{M}_n$ the number of malicious nodes within a shard, and let $\mathscr{T}_n$ represent the total nodes in the shard. Frustum posits that for any shard, particularly the critical L-shard generating new blocks, $\mathscr{M}_n < \frac{1}{3}\mathscr{T}_n$. This limitation is vital for securing a majority consensus among honest nodes, essential for the PBFT algorithm's success within the L-shard.
- **Effective Response to Tampering ($\mathscr{A}_3$).** Assuming a potential tampering event by a malicious leader within a shard, this assumption asserts that a majority of honest nodes, upon detecting tampering via digital signature verification ($\mathscr{V}_a$), will initiate a leader re-election process. This assumption highlights the system's agility and dependability in mitigating threats.

We also define the security properties that the system is designed to uphold as below. These properties are critical as they lay the groundwork for formal security proofs in Section 4.1.:

- **Consensus Integrity ($\mathscr{P}_1$).** The integrity of the consensus process necessitates that each block added to the blockchain, denoted as B_{gen}, results from the accurate execution of the consensus protocol. This implies unanimous agreement on B_{gen}'s contents among all honest nodes.
- **Shard Majority Goodness ($\mathscr{P}_2$).** The security model assumes the majority of nodes ($\mathscr{M}_j$) in any given shard are honest. Symbolically, this means the probability $P(\mathscr{M}_j)$ approaches certainty ($\lim_{n\to\infty} P(\mathscr{M}_j) = 1$) as the network size increases.
- **Resilience to Collusion and Re-sharding Attacks ($\mathscr{P}_3$).** A secure re-sharding mechanism is critical to preventing adversaries from gaining disproportionate control over specific shards during re-sharding events. The system must ensure that it is infeasible for adversaries to predict shard assignments and thus cannot effectively concentrate their malicious efforts.

3.3. A hierarchical sharding architecture

The Frustum system consists of n hosts $N^n = \{N_1, N_2, ..., N_n\}$ scattered at different geographic locations, which connect and communicate with each other via the Internet to form a distributed system. Nodes receive and process transactions and maintain consistent system state as well as data. Each node has a pair of public and private key files for authentication and encrypted message transmission over the network.

Nodes take different roles in the system, including shard leaders and followers. Upon receiving a message from another node, a node feeds back the corresponding response according to the protocol. Eventually, the system provides consensus service to clients if only sufficient nodes work properly.

Frustum adopts a layered structure as shown in Figure 5. Firstly, the system divides all nodes into m shards $S^m = \{S_1, S_2, ..., S_m\}$, which are called F-shards. Each F-shard contains a number of followers and a leader, and all leaders form an L-shard. In each round of consensus, a global leader is selected from the L-shard, which is responsible for generating a block and broadcasting it to the leaders in the F-shards. The F-shard leaders initiate the consensus processes within their respective shards, and finally send consensus results to the global leader, which confirms the termination of the consensus process.

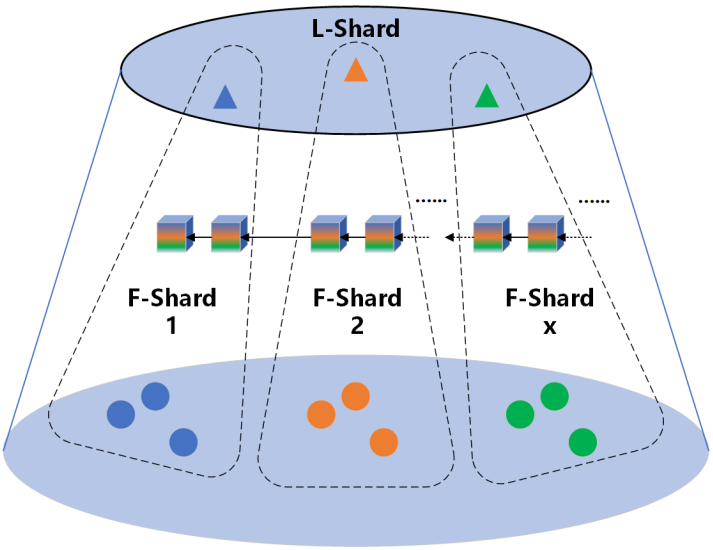


Figure 5. System model of frustum (Triangle: global leaders selected from each F-shard. Circle: other nodes within each F-shard. x: number of F-shards).

3.4. A random re-sharding mechanism

Frustum uses a sharding mechanism similar to Ethereum [26]. In the initialization phase, each node generates a fixed-length address based on its public key, and then all nodes are divided into 2^s shards based on the last s bits of their addresses. The unique aspect of Frustum's sharding mechanism is that it does not re-shard after every round of consensus, as opposed to other sharding systems. Instead, Frustum sets a random round r for shard reorganization. This allows all nodes to perform PoW identity verification and re-shard after r rounds of consensus. We adopt a mechanism similar to *Elastic*'s so that the system can identify each node's identity, where each node needs to find a PoW solution related to *epochRandomness* and node identity information [10]. The *epochRandomness* will be generated in the last step of the previous epoch. After their identities are confirmed, the system divides the nodes into 2^s committees based on the last s bits of the node ID.

As shown in Algorithm 1, this algorithm describes a mechanism for random re-sharding within a distributed system. The algorithm takes as input the total number of nodes (n), the number of shards (m), and the address length (s) used for sharding, and outputs a new shard configuration. Lines 1 to 5 describe the initial sharding function that forms initial shards based on the last (s) bits of the fixed-length addresses generated from each node's public key. Lines 6 and 7 involve selecting a global leader from a specific shard (L-shard), which is assumed to be done through a random selection process. Lines 8 to 12 initiate a consensus mechanism within each shard, led by the shard's leader, and send the consensus result to the global leader for shard consensus. Lines 13 and 14 execute a Proof of Work (PoW) identity verification on nodes. Lines 15 to 30 outline the main algorithm execution process, starting with setting a random round (r) to determine when re-sharding will occur. In each round, if the set round for re-sharding is reached, the shard assignment is updated for nodes that have passed identity verification; otherwise, consensus within the current shard configuration proceeds, preparing for the next round of re-sharding.

By avoiding the need for frequent sharding reorganization, Frustum's approach saves significant resources and time. Additionally, the random re-sharding process makes it challenging for malicious attackers to deploy malicious nodes in one shard during the re-sharding phase. In a non-resharding system, the shard's composition remains fixed and attackers have a prolonged period to observe and analyze the network, which may allow them to strategically position their malicious nodes or activities. Malicious attackers can then deploy malicious

nodes in a specific shard based on the shard structure, making it impossible for that shard to properly validate transactions and thus compromising the security of the system. In a sharding system, malicious attackers will attack when the system is being resharded. With the random-resharding method, a malicious attacker cannot confirm when to launch an attack and has a higher probability of attack failure.

Algorithm 1: Random Re-Sharding in Frustum

Input: A set of nodes $N^n = \{N_1, N_2, \ldots, N_n\}$, number of shards m, last s bits for sharding, fixed-length addresses generated from public keys of nodes

Output: A new sharding configuration after r rounds

```
Function InitialSharding(nodes, s):
    foreach node ∈ nodes do
        address ← GenerateAddress(node.public_key)
        shard_index ← GetLastSBits(address, s)
        Assign node to shard S_shard_index
Function SelectGlobalLeader(L-shard):
    return RandomNode(L-shard)
Function ShardConsensus(F-shards):
    foreach shard ∈ F-shards do
        leader ← shard.leader
        consensus_result ← leader.InitiateConsensus()
        Send consensus_result to global_leader
Function IdentityVerification(node):
    return PoWChallenge(node)
Function Main():
    r ← RandomRound() // Set a random round r for re-sharding
    current_round ← 0
    while true do
        if current_round = r then
            foreach node ∈ N^n do
                if IdentityVerification(node) is valid then
                    Update node's shard assignment
            r ← RandomRound() // Reschedule the next re-sharding round
            current_round ← 0 // Reset the round counter
        else
            F-shards ← InitialSharding(N^n, s)
            L-shard ← FormLShard(F-shards)
            global_leader ← SelectGlobalLeader(L-shard)
            ShardConsensus(F-shards)
            current_round ← current_round + 1
```

3.5. A pipelined consensus protocol

Frustum proceeds in fixed time periods called epochs, and each epoch consists of five phases:

Global Leader Election: Each F-shard elects a leader, and all leaders form an L-shard. Furthermore, the L-shard elects a global leader following a vanilla consensus protocol.

Block Generation: The global leader packages multiple transactions into a block and initiates a consensus process.

Pre-prepare: The global leader sends the block to all L-shard leaders, which then distribute the block to followers within their respective shards. Followers add the new block to the blockchain, and the consensus process for the shard to which the block belongs is initiated, and followers within the shard verify the block.

Prepare: A follower sends verification messages to other nodes within the shard to see whether the transactions are valid.

Commit: After the block is verified, the leader sends a success message to the global leader, and this epoch of consensus ends. If the block validation fails, the leader sends a failure message to the global leader, and then forwards the message to other nodes in the system, and the node will remove the block and all subsequent blocks from the blockchain.

In order to fully utilize resources and improve the parallelism of transaction processing, Frustum designs the above five phases into a pipelined structure as shown in Figure 6.

Figure 7 illustrates an example of committing a block within an epoch. In the following, we will describe the consensus flow of Frustum in detail.

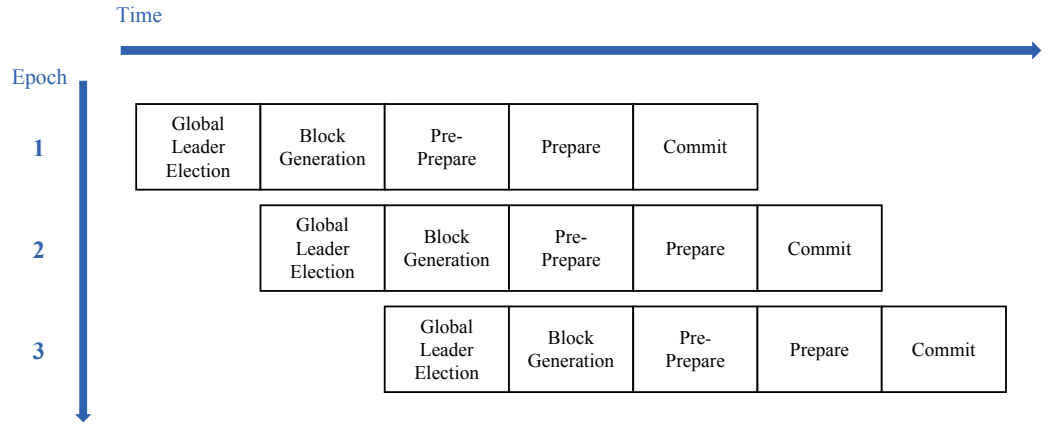


Figure 6. Piplined consensus process of frustum.

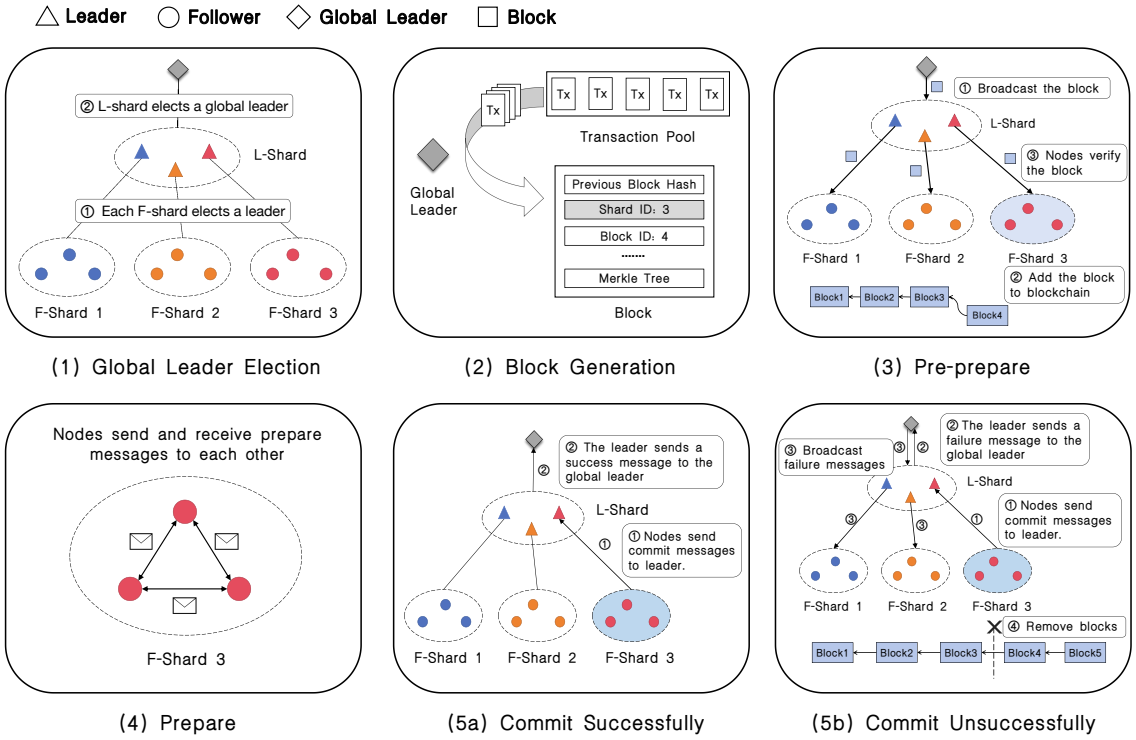


Figure 7. Frustum consensus flow.

3.5.1. Global leader election

Before any transaction processing, a leader within each F-shard should be elected. A random number for each node is assigned by a pseudo-random number generator. The node with the smallest number in each F-shard is the leader of that shard. To ensure randomness, an alternative approach can be adopted by leveraging a verifiable random function (VRF) [27] or a trusted third-party service that provides random number generation for the blockchain [28–30]. A smart contract can serve as a trusted third party to offer random number services, ensuring that the selection of leaders remains unpredictable and secure. By incorporating a VRF or similar trusted third-party mechanism, Frustum can guarantee the randomness and eliminate the possibility of adversaries having advance knowledge of future round leaders. This enhances the overall security and fairness of the leader election process.

A global leader is selected among all F-shard leaders. Besides its role as the leader of its own F-shard, it is also responsible for generating blocks and propagating consensus messages to other L-shard nodes. If the node crashes, the system needs to start a new consensus round for leader selection, which adjourns the transaction processing. Therefore, a stable and reliable global leader is particularly important. Frustum assigns a selection probability to each node based on the node's weight, which is determined by the node's computing power, storage capacity, and past behavior. A node with a higher weight has a higher probability of being selected as the global leader. Once selected, it will send messages to inform other L-shard nodes of its identity.

3.5.2. Block generation

In many blockchain consensus algorithms, multiple nodes may have the right to generate new blocks simultaneously. If different blocks are created at the same time, it may complicate the blockchain with forks [31], where some nodes have inconsistent blockchain states, thereby affecting the stability and reliability of the system. Therefore, Frustum stipulates that only the global leader can package transactions and generate new blocks. By centralizing the block production to the global leader, the Frustum system strategically curtails the incidence of simultaneous block generation, thereby reducing fork occurrences and fortifying the system's stability and dependability.

Specifically, in Frustum, the generation of new blocks is orchestrated by a centrally appointed global leader. Once elected, the global leader proceeds to gather transactions from the pool, which contain the amassed yet unrecorded transactions circulated within the network. These collected transactions are meticulously verified by the leader, confirming the authenticity of digital signatures, the absence of double-spending, and overall adherence to the network's established protocols. Upon successful verification, the transactions are adeptly structured into a Merkle tree [32]—a binary tree where each leaf is a transaction's hash and each non-leaf node is the concatenated hash of its children. This configuration is not only space-efficient but also secures the integrity of transaction verification within a block.

Utilizing this Merkle tree, the global leader then crafts a block that encapsulates the tree or its root hash, the transaction list, and fundamental block metadata like block height, which identifies the block's sequential order within the chain, and the timestamp, marking the precise moment the block was constituted.

This newly created block is then disseminated across the network, whereupon receiving nodes engage in validation checks against the Merkle root and the block's compliance with the blockchain's protocol. Successful validation leads to the incorporation of the block into the blockchain, a crucial step that concurrently updates the network's state with the latest transactions and acknowledges the new block height uniformly across the network.

3.5.3. Pre-prepare

After the global leader generates a block, it sends the new block to all L-shard nodes, which then forwards the block to other nodes in their respective F-shards, and further on all nodes add the block to their local blockchain storage.

An F-shard leader needs to determine whether the block belongs to its own shard, and if so, it sends a pre-prepare message to its followers to initiate the verification process. The followers receive the pre-prepare message and validate the content of the message, i.e., the new block to be committed.

A hash-based assignment method is effective for an F-shard leader to determine if a block belongs to its shard. Essentially, the process involves applying a predetermined hash function to the block's header or a designated sharding field within the block to produce a hash value. The leader then compares this hash value with the hash ranges assigned to its shard to verify whether the block falls within its jurisdiction. If the block's hash value lies within the leader's range, it is recognized as part of the F-shard, prompting the leader to initiate the verification process by sending out a pre-prepare message to its followers. This method ensures a straightforward and secure mechanism for block assignment in a distributed blockchain network.

3.5.4. Prepare

After the block is verified, the follower broadcasts a prepare message to other members within its shard. If a node receives correct prepare messages from more than $2s/3+1$ different nodes, where s is the total number of nodes in the corresponding shard, the next phase starts.

3.5.5. Commit

During the commit phase, the node will send commit messages to other nodes within its shard. When the number of commit messages received by a node is higher than $2s/3+1$, it sends a confirmation message to the leader of the corresponding shard.

In a sharded blockchain architecture, the integrity of the commit phase is of paramount importance, wherein the shard leaders are tasked with disseminating a success or failure message contingent upon achieving a consensus. This message is methodically structured to include several critical components:

- MessageType: A binary flag that unequivocally indicates the nature of the message, discerning between a success and a failure outcome.
- ShardId: A distinctive identifier that demarcates the originating shard, ensuring the proper attribution of the message to its source.
- BlockMetadata: Comprising the block's hash and height, supplemented by additional metadata to uniquely characterize the block under consideration.
- VoteCount: An enumeration of the confirmatory messages received, serving as an unambiguous metric of consensus.
- Timestamp: A temporal marker denoting the message's issuance, crucial for synchronicity and delay analysis across the network.
- DigitalSignature: An authentication schema that corroborates the message's provenance and safeguards against alteration, assured by the shard leader's cryptographic signature.

Once the leader receives confirmation messages from more than $s/3+1$ different followers, which means the shard reaches a consensus to submit the block. If $s/3+1$ confirmation messages are not received within the specified time, then the block cannot be successfully submitted in this round of consensus, it may be discarded or wait for another round. The leader sends a verification failure message to the global leader, which then forwards the failure message to other nodes in the system. The nodes remove the block and all subsequent blocks

from the blockchain and terminate the consensus process for those blocks, a.k.a *flushing*. The specified time here refers to the longest transaction processing time, which can be obtained by statistically analyzing the processing time of a large number of transactions.

To fortify the message transmission process against the potential dissemination of misleading information by a shard leader to the global leader. We have employed three mechanisms: signature validation, cross-validation, and penalization. First, each confirmation missive is mandated to be signed by its sender and the collection of these authentication tokens is then integrated into the shard leader's message, enabling the global leader to perform a verification of the signatures to validate the legitimacy of the received confirmations. Second, the global leader is not solely dependent upon the shard leader's testimony. It engages in corroboratory data collection from a multiplicity of nodes, particularly in cases where the information relayed by the shard leader is indicative of a failure. Finally, incentive alignment and penalty mechanisms ensure that shard leaders are punished if they are discovered to have propagated false assertions. Such consequences involve the reduction of staking privileges or the removal of the leader from their incumbency, thereby fostering an environment conducive to veracious conduct. The amalgamation of these measures into the consensus algorithm augments the resilience of the commit phase, diminishing the propensity for malfeasance while amplifying inter-shard communicative reliability.

4. System analysis

4.1. Security analysis

In this section, we provide security analysis for how Frustum prevents potential threats and works securely. We first formalize the security definition of Frustum as follows: Consider a set of nodes with equal computational power, where a fraction is controlled by Byzantine adversaries. The system parameters include: n, the total number of nodes we aim to generate in one resharding instance; f, the number of malicious nodes in the shard; $f' = \frac{1}{4}$, the fraction of computational power controlled by malicious users; c, the size of each shard; and m, the number of shards. We assume that Byzantine adversaries can act arbitrarily, including sending false or misleading information or causing unpredictable damage within the system. Notably, they can engage in:

- **Voting Manipulation:** In a distributed system requiring voting, Byzantine adversaries may attempt to unduly influence the voting process, for instance, by casting fraudulent votes to alter the outcome.
- **Impersonation:** Byzantine adversaries have the capability to impersonate other legitimate nodes, especially leader nodes, sending false messages to misguide the system or other nodes.
- **Collusion Attacks:** Attackers may strategically position malicious nodes within the target shard and collaborate to create greater disruption, such as executing double-spending attacks.

The adversary considered by the protocol is adaptive. Adversaries adjust their strategies based on changes within the system, notably during re-sharding events which suggest the need for adversaries to adapt to new configurations of nodes. To prevent the success of these attacks, we define a system as secure if it includes the following safety properties:

1. A supermajority consensus can be achieved when the number of malicious nodes in a shard is less than 1/3 of the total nodes in the respective shard.
2. Honest nodes dominate in all the shards that are generated. Given a control parameter λ, we can deduce a threshold n_0 such that for all $n > n_0$, the probability of the system maintaining its majority goodness approaches 1.
3. Upon the occurrence of a re-sharding, we guarantee that it is challenging for the adversary to strategically locate and launch a collusion attack against target nodes.

First we prove that:

Theorem 1. *To achieve the security of the system, Frustum requires that the number of malicious nodes in each shard is less than 1/3 of the total number of nodes in the corresponding shard, especially the L-shard.*

Proof. The L-shard is a very important part of the system and is responsible for generating new blocks. Frustum uses the consensus algorithm of PBFT within the L-shard to reach a consensus on new blocks. The system needs a majority agreement of honest nodes ($(n-f)/2$) to reach consensus, and in addition, all malicious nodes may oppose, so these majority of honest nodes need to outnumber malicious nodes to avoid malicious nodes winning, which means $(n-f)/2 > f$. From this, we can infer that the number of malicious nodes is less than 1/3 of the total number.

If the leader of the shard is a malicious node that tampers with the original request before initiating consensus. Thanks to digital signature, the client will send the original request with a digital signature, and the honest node will verify the authenticity of the request by a digital signature, and when most of the honest nodes find the leader tampered message, it will trigger the view switch, i.e., re-elect the leader.

Theorem 2. *Good Majority in Shards. For every sufficiently large integer $n \geq n_0$: among the first n identities created, at most $n_0/3-1$ are controlled by the adversary w.h.p.*

Proof. If all the users start at the same time, each solution generated has a probability $1-f'$ of being taken by the honest nodes. Now, let X_i be an indicator random variable which takes the value one if the i-th identity is generated by an honest node. Let $X = \sum_{i=1}^{n_0} X_i$. Then, X follows a binomial distribution.

Thus, we have:

$$\Pr(X \leq 2n_0/3) = \sum_{k=0}^{2n_0/3} \Pr[X=k] = \sum_{k=0}^{2n_0/3} \binom{n_0}{k} f'^k (1-f')^{n_0-k}.$$

This probability decreases exponentially in n_0. Given a security parameter λ, we can find n_0 such that $\Pr[X \leq \frac{2n_0}{3}] \leq 2^{-\lambda}$, for all $n \geq n_0$. The committee size is at least n_0 to guarantee that the fraction of malicious members in a committee is bounded by $1/3$, with regard to the security parameter λ. The value of n_0 depends on the security parameter λ. For example, if $\lambda = 20$, or the probability that something bad happens is once every 1 million epochs, we have $n_0 \approx 600$.

Theorem 3. *Good randomness. Due to the randomness in re-sharding and the integrity of the consensus protocol, their ability to attack the specific shard is limited and the security in re-sharding is guaranteed.*

Proof. Frustum's sharding mechanism employs randomness in two key ways to ensure the security and functionality of the sharding process. Firstly, Frustum assigns nodes to shards based on the deterministic output of a hash function applied to their public keys. The system leverages the pseudo-random nature of hash functions to evenly distribute nodes across shards by utilizing the last s bits of the ID of nodes that have been verified through PoW, assigning them to one of 2^s shards. Regarding the unpredictable and consistent randomness of PoW, we employ a PoW verification mechanism consistent with Elastico. The proof of the good randomness of this mechanism has been verified in [10]. Secondly, Frustum introduces a distinctive feature by eschewing a fixed re-sharding schedule. Instead, the network selects a random source of randomness, such as a verifiable random function (VRF), to determine the timing of re-sharding. By avoiding predictable re-sharding after each consensus round, Frustum effectively thwarts attackers' ability to anticipate when re-sharding will take place. These unpredictability significantly hampers the attacker's capacity to strategically position malicious nodes within a targeted shard, bolstering the system's security.

In essence, Frustum's security is partly based on the principle of entropy: randomness increases uncertainty for attackers, making it statistically improbable for them to gain control over a specific shard or predict the system's re-sharding schedule. This randomness is a critical element of the system's design for maintaining robust security against coordinated attacks on its sharded blockchain architecture.

4.2. Performance analysis

4.2.1. Confirmation latency

In Frustum, confirmation latency refers to the time it takes for a transaction to go from being sent to being confirmed. According to the consensus process described in Section 3, transactions are first sent from the client to the consensus node with time duration, T_{Req}. The consensus node then caches it in the local transaction pool and waits for $T_{PoolWait}$ to get the right to pack the transaction into a new block. When the consensus node obtains the right to generate a new block, it starts a new round to reach consensus after $T_{Consensus}$. Finally, the consensus node spends T_{Ack} to reply to the client that the transaction has been committed. In these four different times, T_{Req} and T_{Ack} have a relatively fixed value, they may only be relevant to the network from the client to the consensus node, which is bandwidth dependent. $T_{Consensus}$ refers to the time required to reach consensus among consensus nodes. The normal case is the time overhead of three rounds of communication, and if the consensus fails, the transactions in the block go back to the transaction pool and wait for the new round. So $T_{PoolWait}$ is a random value with an upper bound, it may be very short when the transaction is packed into a block immediately after arriving at the transaction pool, or long enough with multi $T_{PoolWait}$. The confirmation latency l is shown in the following formula 1.

$$l = T_{Req} + T_{PoolWait} + T_{Consensus} + T_{Ack} \tag{1}$$

The network delay in the Frustum blockchain system is similar to a partially synchronous network model (PDFT). The partially synchronous model accounts for periods of synchrony (with known bounds) and asynchrony (randomness), which aligns with the Frustum system's operation. While T_{Req} and T_{Ack} provide the periods where network behavior is predictable and bounded, $T_{PoolWait}$ introduces variability that can't be predicted ahead of time, although it does have an upper bound. $T_{Consensus}$ also has a typical bounded time under normal conditions. This combination of fixed and variable components, along with the presence of upper bounds, is characteristic of a partially synchronous network model.

4.2.2. Storage consumption

Unlike a full sharding blockchain, all nodes in Frustum maintain the same single blockchain. The effective data storage capacity is equal to the node with the smallest storage capacity in the system. This drawback has the better fault tolerance of the system, even if some of the nodes are offline, the system can still work normally and can continue to serve correctly after recovery. Liveness analysis gives the exact number of offline nodes and shards that can be tolerated. A complete sharding blockchain cannot tolerate the nodes of any 1/3 of the shard being offline, which will result in the transactions of the relevant shard not being committed and having to wait for recovery. Due to the fact that each shard stores different blockchain data, a complete sharding blockchain enables fuller use of storage, the more shards there are, the more storage resources can be used.

5. Evaluation

In this section, the Implementation of a prototype of Frustum is presented. Then we describe the experimental environment and dataset. At last, the comparative results with two state-of-

the-art algorithms are shown.

5.1. Implementation

We implement a prototype of Frustum and its comparative algorithms in Golang [33] to evaluate their performance. By using concurrency models in Golang, it is easy to execute a large number of concurrent tasks, enabling the creation of multiple blockchain nodes locally and simulating the pipelined consensus process. We also implement the necessary modules for experimental evaluation, such as blockchain data storage, transaction pools, and network communication.

In the implementation, a client has a corresponding F-shard leader and continuously sends transaction requests to it. The node caches them in its transaction pool. Once the node becomes a global leader and obtains the right to issue blocks, it packs the transactions from the transaction pool to generate new blocks and then initiates the consensus process.

5.2. Methodology

5.2.1. Testbed

We evaluate the Frustum prototype on a workstation with 20 CPU cores (@3.7GHz) and 125GB memory.

Each experiment in this section executes a consensus algorithm on 512 blocks and the system has 1024 nodes by default. Only the statistics gathered during the stable phase of the system operation is taken into account, i.e. that of the warming up and cooling down phases is excluded.

5.2.2. Transaction dataset

We employ a real blockchain transaction trace, named Ethereum transactions, extracted from XBlock-ETH [24]. The dataset comes from the real Ether transaction data pulled up using the XBlock-ETH tool, with block heights ranging from 0 to 14,499,999 and a total number of transactions of 1,524,325,672. We randomly select 500,000 transactions from it as experimental data, and fine-tune their data structure to fit our experiments.

The transaction dataset also contains cross-shard transactions for full sharding systems that maintain a multi-chain structure, such as RapidChain [9]. As shown in Figure 8, the proportion of cross-shard transactions increases as the number of shards increases. When the shard number is 16, the proportion of cross-shard transactions exceeds 90%, which means that more than 90% of transactions cannot be processed on the local node and require inter-shard communication to ensure correct submission.

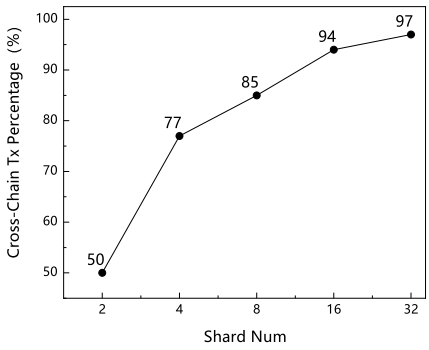


Figure 8. Percentage of cross-shard transactions in the Ethereum dataset with different shard number.

5.2.3. Baselines

Frustum is a shard-based, single-chain blockchain consensus protocol, therefore we compare it against two state-of-the-art blockchain systems, Elastico [10] and RapidChain [9], which are partial sharding and full sharding blockchain systems respectively.

For fairness, the classical PBFT consensus algorithm is adopted for all intra-shard and inter-shard consensus progress, which avoids any impact of different underlying consensus algorithms.

5.3. Results

The experimental results are shown and analyzed in this section.

5.3.1. Choice of block size

To determine a reasonable block size, we measure the throughput and latency of Frustum with the number of transactions contained in the block changing from 16 to 128.

As shown in Figure 9, with an increasing number of transactions in a block, both the transaction throughput and commit latency show an upward trend. In order to ensure that the system can meet the requirements of mainstream payment systems for low latency while having the highest possible throughput, we set the number of block transactions to 64 and further analyse the system performance on this basis.

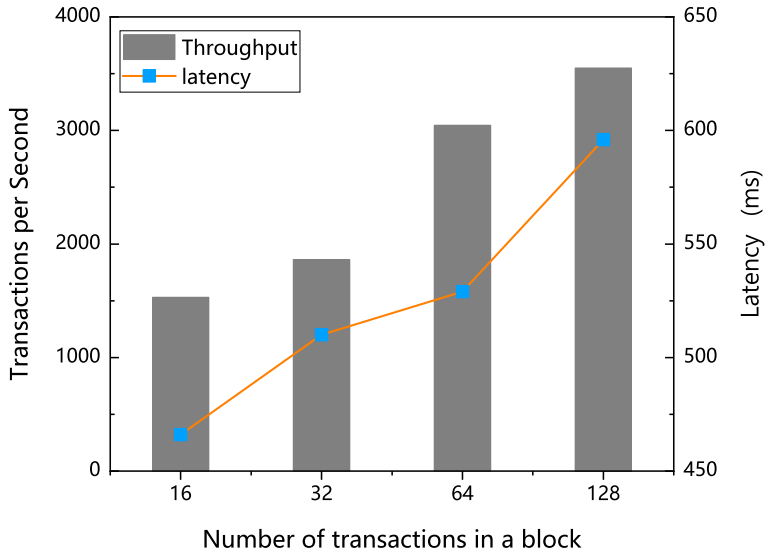


Figure 9. The throughput and latency of Frustum change with the number of block transactions.

5.3.2. Influence of pipelining

This experiment presents the impact of using a pipelining structure on transaction throughput as the number of shards increases. The comparative algorithm uses no pipelining but shares the same consensus process as Frustum.

Figure 10 shows the changes in throughput of the comparison algorithm and Frustum as the number of shards increases from 2 to 16. More shards mean increasing in parallelism and that more transactions can be processed simultaneously, thus the throughput of both algorithms increases. At the same time, with all shard numbers, the performance of Frustum with a

pipelined structure is always superior to that of the comparison algorithm without pipelining. For example, with 16 shards, the throughput of the algorithms without and with pipelining is 3383 tx/sec and 4699 tx/sec respectively, increasing by 39%. It can be seen that pipelining significantly improves transaction processing parallelism and optimizes system performance.

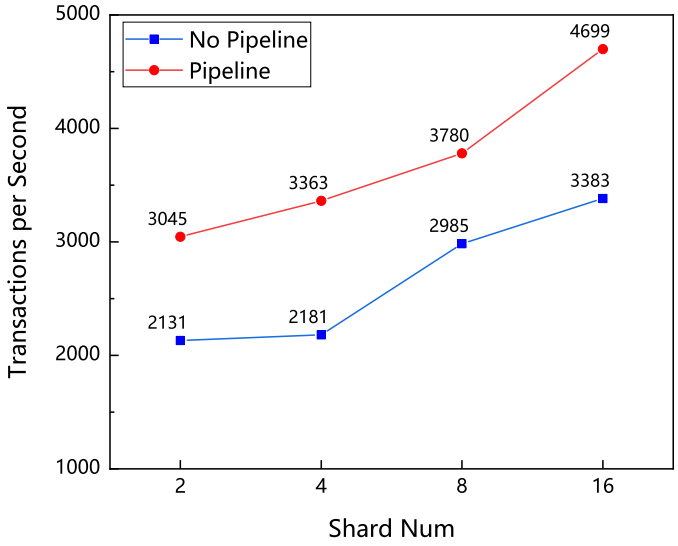


Figure 10. The throughput changes with the number of shards w/o a pipeline structure.

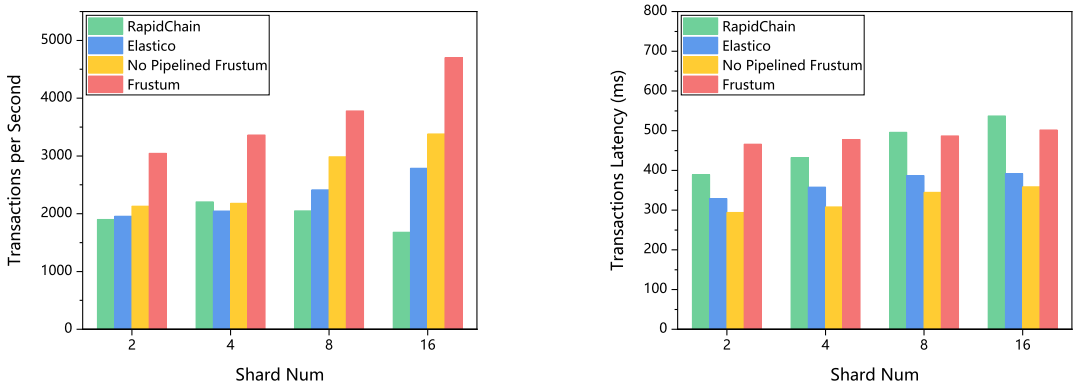
5.3.3. Influence of shard number

Frustum is a sharding-based consensus protocol, therefore We compare throughput and latency of Frustum with Elastico and RapidChain as the number of shards varies.

As shown in Figure 11a, the transaction throughput of Frustum w/o a pipelined structure always outperforms the other two algorithms. When the shard number is 16, the throughput of RapidChain, Elastico, and Frustum is 1680 tx/sec, 2787 tex/sec, and 4699 tx/sec respectively. Frustum is 2.79 times and 1.68 times faster. The throughput of both Frustum and Elastico increases as the number of shards increases, but Elastico increases relatively slower. Elastico requires that each round of consensus involves reorganizing the shards, and the updated shard states are sent to every node. The frequent shard reorganizing increases the computational and communication costs, leading to a decrease in system performance. Additionally, each block needs to be further validated by the final committee, which increases the processing time for transactions, thereby affecting the system's throughput. RapidChain is a consensus algorithm with a multi-chain structure. The proportion of cross-shard transactions increases with the shard number. RapidChain splits cross-shard transactions into multiple sub-transactions and indirectly verifies the original transaction by proving the validity of the sub-transactions. As shown in Figure 8, even with only 4 shards, the majority of transactions should be split. Substantial cross-shard processing significantly increases the average latency of transactions and thus cancels out the benefit of sharding, or even worsens the transaction throughput. Maintaining a single blockchain across shards guarantees the scalability of Frustum in terms of transaction throughput.

Figure 11b shows how the transaction commit latency of the four algorithms changes as the number of shards increases. Frustum without a pipeline structure always has a lower latency than Elastico and RapidChain, while pipelined Frustum's latency increases compared to others. This is because the pipeline structure requires additional pipeline communication to synchronize the process, resulting in more processing time for one transaction. Therefore, for

some latency-sensitive applications, No pipelined Frustum can be applied as an alternative to the Frustum.



(a) Throughput **(b)** Transaction Commit Latency

Figure 11. The throughput and latency of Frustum, Elastico and RapidChain changes with the number of shards.

5.3.4. Evaluation summary

In a nutshell, from a system reliability perspective, a full sharding system splits data into individual shards, and if a shard becomes unavailable, some cross-shard transactions cannot be committed until the shard is restored. Therefore, a full sharding system sacrifices system reliability to increase throughput. Partial sharding systems, such as Frustum and Elastico, maintain a consistent single-chain structure across all nodes, and the failure of one shard does not affect cross-shard and other transaction submissions. Consequently, in cross-shard transaction processing, Frustum is faster than Rapidchain. As the number of shards increases, there are more cross-shard transactions in the network, requiring Rapidchain to frequently split cross-shard transactions and wait for all sub-transactions to confirm. Hence, as the number of shards in Rapidchain increases, throughput will also decrease significantly.

Experiments also demonstrated that Frustum with a pipeline structure performs better than those executed in serial loops. Additionally, for latency-insensitive scenarios, Frustum without a pipeline structure can be applied as an alternative to the Frustum.

6. Related work

In this section, we review three categories of blockchain consensus systems, namely no sharding system, partial sharding system, and full sharing system.

6.1. No sharding system

Bitcoin [1] is the most famous no sharding system, using Proof of Work (PoW) [34] as a consensus algorithm. In a no sharding blockchain system, all nodes in the network process and store every transaction [35], which means that as the number of nodes and transactions increase, the system becomes slower and less efficient. This lead to longer confirmation times and poor scalability. Furthermore, a no sharding system can also be more vulnerable to attacks such as 51% attacks [36], where a single entity controls more than half of the network's computing power and can manipulate the blockchain's history.

6.2. *Partial sharding system*

Elastico [10] is the first public sharding-based blockchain system in which each shard maintains its own independent blockchain and is responsible for processing a subset of the overall transactions. This allows for more efficient processing and increased transaction throughput. Additionally, Elastico achieves consensus based on PBFT, which helps to maintain the security of the system and prevent malicious behavior. Elastico only implements network and transaction sharding, but does not achieve state sharding in storage and communication, thus Elastico is a partial sharding system.

Although Elastico improves the throughput and transaction latency compared to Bitcoin, it still has several drawbacks. On the one hand, Elastico requires all nodes solving PoWs to reconstruct identities and rebuild all shards in every epoch, this results in additional overhead and limits the system performance [9]. On the other hand, the block needs to be validated by the final shard before it can be stored in the global blockchain, which adds complexity to the consensus process and increases the average submission latency of transactions.

6.3. *Full sharding system*

Another sharding system is full sharding, where transactions, network, and data storage states are all sharded. Blockchain systems like Monoxide [37], OmniLedger [8] and Rapidchain [9]are full sharding systems, each shard maintains different blockchain data independently, also called multi-chain.

Omniledger is the first full sharding blockchain platform that aims to provide high scalability and security while supporting cross-shard transactions. To enable cross-shard transactions, Omniledger uses a technique called "atomic cross-shard transactions," which ensures that either all the involved shards commit the transaction or none of them do. This is done by first locking the assets on the source shard and then transferring them to the destination shard through a series of transactions that are guaranteed to be atomic. If any of the transactions fail, the entire process is rolled back to its initial state. RapidChain adopts a similar method, it splits the original cross-shard transaction into multiple sub-transactions for validation, and the confirmation of sub-transactions indirectly indicates the validity of the original transaction. The difference between OmniLedger and Rapidchain is whether the client needs to be involved in the whole process of the cross-shard transaction. Rapidchain chooses a light client, it simply submits a transaction to nearby consensus nodes without having to know the entire network topology and engages in the cumbersome processing mechanism of cross-shard transactions.

Although full sharding has improved the scalability of the blockchain system, as the percentage of cross-shard transaction increases, the network is flooded with inter-shard communication, and each cross-shard transaction requires additional processing, it may affect the system performance and undermine the benefits of parallelization from full sharding.

7. Conclusion

In this article, we propose Frustum, a sharding-based blockchain consensus protocol using a pipelined structure. Frustum maintains a globally consistent single blockchain, avoiding additional communication costs caused by cross-shard transactions. Frustum also uses a layered structure to reduce communication overhead and the average time for processing transactions, thus increasing system throughput. Furthermore, Frustum divides the consensus process into multiple stages, and a pipeline structure is adopted to allow parallel execution of each stage. Finally, our empirical evaluation demonstrates that Frustum showing better performance than previous work.

Conflicts of interests

The authors declared that they have no conflicts of interests.

Authors' contribution

Yukun Xu proposed the preliminary ideas. Wenhan Wu carried out the model definition and theoretical analysis. Yukun Xu and Wenhan Wu collaborated in designing and analyzing the algorithms and contributed equally to this work. Yili Gong and Kanye Ye Wang conducted experiments and tests. Chuang Hu and Dazhao Cheng participated in the research of related work. All authors collectively contributed to the writing part and made contributions to the paper.

References

[1] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Bus. Rev.* 2008, 21260.

[2] Dai HN, Zheng Z, Zhang Y. Blockchain for Internet of Things: A survey. *IEEE Internet Things J.* 2019, 6(5):8076–8094.

[3] Khanna R. A review paper on characteristics of blockchain. *AJMR* 2021, 10(10):1036–1040.

[4] Bamakan SMH, Motavali A, Bondarti AB. A survey of blockchain consensus algorithms performance evaluation criteria. *Expert Syst. Appl.* 2020, 154:113385.

[5] Gemeliarana IGAK, Sari RF. Evaluation of proof of work (POW) blockchains security network on selfish mining. In *2018 International seminar on research of information technology and intelligent systems (ISRITI)*, Yogyakarta, Indonesia, November 21–22, 2018, pp. 126–130.

[6] Huang H, Peng X, Zhan J, Zhang S, Lin Y, *et al.* Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, London, United Kingdom, May 02-05, 2022, pp. 1968–1977.

[7] Xi J, Zou S, Xu G, Guo Y, Lu Y, *et al.* A comprehensive survey on sharding in blockchains. *Mob. Inf. Syst.* 2021, 2021:1–22.

[8] Kokoris-Kogias E, Jovanovic P, Gasser L, Gailly N, Syta E, *et al.* Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 20–24, 2018, pp. 583–598.

[9] Zamani M, Movahedi M, Raykova M. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, Toronto, Canada, October 15–19, 2018, pp. 931–948.

[10] Luu L, Narayanan V, Zheng C, Baweja K, Gilbert S, *et al.* A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, Vienna, Austria, October 24–28, 2016, pp. 17–30.

[11] The ZILILLIQA Team. The zilliqa technical whitepaper. *Retrieved September* 2017, 16:2019.

[12] Lamport L, Shostak R, Pease M. The Byzantine generals problem. In *Concurrency: the works of leslie lamport*, New York: Association for Computing Machinery, 2019, pp. 203–226.

[13] Preneel B. Secure Information Networks: Communications and Multimedia Security. In *Proceedings of the IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS'99)*, Leuven, Belgium, September 20–21, 1999.

[14] Lu Y. Blockchain: A survey on functions, applications and open issues. *J. Ind. Manag.* 2018, 3(4):1850015.

[15] Liu L, Xu B. Research on information security technology based on blockchain. In *2018 IEEE 3rd international conference on cloud computing and big data analysis (ICCCBDA)*, Chengdu, China, April 20–22, 2018, pp. 380–384.
[16] Lemieux VL. Trusting records: is Blockchain technology the answer? *Rec. Manag. J.* 2016, 26(2):110–139.
[17] Zaghloul E, Li T, Mutka MW, Ren J. Bitcoin and blockchain: Security and privacy. *IEEE Internet Things J.* 2020, 7(10):10288–10313.
[18] Gervais A, Karame GO, Wüst K, Glykantzis V, Ritzdorf H, *et al.* On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, Vienna, Austria, October 24–28, 2016, pp. 3–16.
[19] King S, Nadal S. PPCoin: Peer-to-peer crypto-currency with proof-of-stake. Available: https://www.peercoin.net/whitepapers/peercoin-paper.pdf (accessed on 15 December 2023), 2012,.
[20] Larimer D. Delegated proof-of-stake (dpos). *Bitshare whitepaper* 2014, 81:85.
[21] Ruoti S, Kaiser B, Yerukhimovich A, Clark J, Cunningham R. Blockchain technology: what is it good for? *Commun. ACM* 2019, 63(1):46–53.
[22] Castro M, Liskov B. Practical byzantine fault tolerance. In *OSDI*, New Orleans, Louisiana, USA, February 22-25, 1999, 99, pp. 173–186.
[23] Latifa ER, Omar A. Blockchain: Bitcoin wallet cryptography security, challenges and countermeasures. *Int. J. Electron. Commer.* 2017, 22(3):1–29.
[24] Zheng P, Zheng Z, Wu J, Dai HN. Xblock-eth: Extracting and exploring blockchain data from ethereum. *IEEE OJ-CS* 2020, 1:95–106.
[25] Dolev D, Yao A. On the security of public key protocols. *IEEE Trans. Inf. Theory* 1983, 29(2):198–208.
[26] Wood G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 2014, 151(2014):1–32.
[27] Micali S, Rabin M, Vadhan S. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, New York, NY, USA, October 17–19, 1999, pp. 120–130.
[28] Bugday A, Ozsoy A, Sever H. Securing blockchain shards by using learning based reputation and verifiable random functions. In *2019 International Symposium on Networks, Computers and Communications (ISNCC)*, Istanbul, Turkey, June 18–20, 2019, pp. 1–4.
[29] Kate A, Mangipudi EV, Maradana S, Mukherjee P. FlexiRand: Output Private (Distributed) VRFs and Application to Blockchains. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, Copenhagen, Denmark, November 26–30, 2023, pp. 1776–1790.
[30] Chatterjee K, Goharshady AK, Pourdamghani A. Probabilistic smart contracts: Secure randomness on the blockchain. In *2019 IEEE international conference on blockchain and cryptocurrency (ICBC)*, Seoul, Korea (South), May 14–17, 2019, pp. 403–412.
[31] Goldberg I, Moore T. *Financial Cryptography and Data Security*. 2019.
[32] Merkle RC. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87*, Santa Barbara, CA, USA, August 16–20, 1987, pp. 369–378.
[33] Westrup E, Pettersson F. Using the go programming language in practice. *Master's Thesis,* Lund University, 2014.
[34] Jakobsson M, Juels A. Proofs of work and bread pudding protocols. In *Secure Information Networks: Communications and Multimedia Security IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS'99)*, Leuven, Belgium, September 20–21, 1999, pp. 258–272.
[35] Dai M, Zhang S, Wang H, Jin S. A low storage room requirement framework for

distributed ledger in blockchain. *IEEE Access* 2018, 6:22970–22975.

[36] Ye C, Li G, Cai H, Gu Y, Fukuda A. Analysis of security in blockchain: Case study in 51%-attack detecting. In *2018 5th International conference on dependable systems and their applications (DSA)*, Dalian, China, September 22–23, 2018, pp. 15–24.

[37] Wang J, Wang H. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th USENIX symposium on networked systems design and implementation (NSDI 19)*, Boston, MA, USA, February 25–28, 2019, pp. 95–112.