A Reliable Storage Partition for Permissioned Blockchain

Xiaodong Qi, Zhao Zhang, Cheqing Jin, and Aoying Zhou

Abstract—The full-replication data storage mechanism, as commonly utilized in existing blockchains, is the barrier to the system's scalability, since it retains a copy of entire blockchain at each node so that the overall storage consumption per block is $O(n)$ with n participants. Yet another drawback is that this mechanism may limit the throughput in permissioned blockchain. Moreover, due to the existence of Byzantine nodes, existing partitioning methods, though widely adopted in distributed systems for decades, cannot suit for blockchain systems directly, so that it is critical to devise new storage mechanism for blockchain systems. This article proposes a novel storage engine, called BFT-Store, to enhance storage scalability by integrating erasure coding with Byzantine Fault Tolerance (BFT) consensus protocol. The first property of BFT-store is that the storage consumption per block can be reduced to $O(1)$ for the first time, which enlarges overall storage capability when more nodes attend the blockchain. Second, we design an efficient online re-encoding protocol for storage scale-out and a hybrid replication scheme to enhance reading performance. Analysis in theory and extensive experimental results illustrate the scalability, availability and efficiency of BFT-Store via the implementation in an open-source permissioned blockchain Tendermint.

Index Terms—Blockchain, erasure coding, Byzantine fault tolerance, storage scalability

1 Introduction

Blockchain is a distributed append-only ledger maintained by all nodes in an untrustworthy environment. In order to avoid being tampered by malicious nodes, blocks in the blockchain are chained by cryptograph hash values in the block headers. In general, each node holds a complete copy of block data, and consensus protocol, such as PoW [1] in permissionless blockchain and Practical Byzantine Fault Tolerance (PBFT) [2] in permissioned blockchain, ensures the data consistency among whole network. Thus the overall storage consumption per block is $O(n)$, where n is the number of nodes. However, with this full-replication manner, each node has to devote huge storage space to preserving blocks, e.g., the total amount of block data of Bitcoin is over 200 GB right now and the Ethereum network generates about 0.2 GB data every day.[1] Permissioned blockchains are designed to support applications beyond cryptocurrency such as finance management, banking and insurance, where participants are authenticated and do not fully trust each other and everyone has the potential to do malicious things for its own benefit. Commonly, permissioned blockchains adopt PBFT protocol to achieve higher transaction throughput, thus the amount of block data will break through the storage capacity of single node. A natural, but challenging issue arises to be: *can we lower the storage consumption but still be capable of recovering the whole blockchain at any time?*

To address above, some techniques are introduced, such as light-weight client [1], [3] and lightning network [4]. Each light-weight client just saves block headers rather than entire blockchain to verify block bodies received from full nodes. However, this technique just reduces the overhead of client while the full node still reserves a complete copy of all block data. The basic idea of lightning network is to restrain the volume of block data by gathering multiple micropayments between two accounts and rarely uploading the final balances to blockchain. Although this approach eases the stress of storage for each node, it does not subvert the full-replication mechanism fundamentally. The sharding technique [5], [6] divides entire network into multiple groups, each of which stores a part of data. However, in same group, each node retains a full copy of partial block data as well. Furthermore, due to the scale of blockchain becomes smaller, the throughput of each group increases correspondingly and blocks are generated at a faster speed, which adds the storage overhead to each node.

A straightforward concern is whether the partitioning manner (e.g., hash or range partitioning), which is widely adopted in distributed systems, suits for blockchain systems or not. In this mechanism, the original data is divided into a number of partitions and dispatched to all nodes. Particularly, for high availability, each partition is replicated for several times (e.g., 3 in Hadoop [7]) to tolerate the fault of single node. However, this mechanism cannot suit for blockchain due to the existence of the malicious nodes who may keep silent or tamper data deliberately. For example, if all replicas of a block happen to be dispatched to malicious

1. https://www.statista.com/

• *Xiaodong Qi, Cheqing Jin, and Aoying Zhou are with the School of Data Science and Engineering, East China Normal University, Shanghai 200062, China. E-mail: xdqi@stu.ecnu.edu.cn, {cqjin, ayzhou}@dase.ecnu.edu.cn.*
• *Zhao Zhang is with the School of Data Science and Engineering, East China Normal University, Shanghai 200062, China, and also with the Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541004, China. E-mail: zhzhang@dase.ecnu.edu.cn.*

Manuscript received 25 Nov. 2019; revised 18 June 2020; accepted 13 July 2020. Date of publication 29 July 2020; date of current version 7 Dec. 2020.
(Corresponding author: Zhao Zhang.)
Recommended for acceptance by J. Xu.
Digital Object Identifier no. 10.1109/TKDE.2020.3012668

nodes, this block may be tempered, or even lost. To ensure each block can be recovered, the only way is to generate more replicas, at least more than the potential malicious nodes. However, it will be too expensive as well.

Alternatively, *erasure coding*(EC) provides significantly lower storage overhead than multiple-replication manner at the same fault tolerance level [8]. In a nutshell, EC transforms original blocks into a set of coded blocks, namely "chunks", such that any subset with sufficient available chunks can reconstruct the original data. However, EC achieves storage efficiency at the cost of high repair penalty. Specifically, the repair of a single failed chunk (either lost or unavailable) needs to read multiple available chunks for reconstruction. In other words, it reads more available data than the actual amount of failed data. Besides, although existing systems can handle omission node failures (e.g., fail-stop) and won't taint other nodes, they cannot deal with Byzantine failures [9]. A node will reconstruct incorrect data based on the invalid chunks sent by Byzantine nodes.

Intuitively, it is feasible to devise an EC-based storage partitioning for blockchain systems. It is common that existing Byzantine fault tolerance consensus protocols require the number of malicious nodes f below a certain rate of all nodes, such as no more than $1/3$ nodes are malicious for PBFT [2], e.g., $n \geq 3f+1$. Take the PBFT protocol as an example and consider the case that $n = 3f+1$ where at least $2f+1$ non-faulty (correct) nodes exist. We can divide a block into $2f+1$ *sub-blocks* and encode them into $n = 3f+1$ chunks with f checksum blocks, termed as "parities", for redundancy via *Reed-Solomon* [10], a widely used EC. Then the $3f+1$ chunks are distributed over $3f+1$ nodes, thereby at least $2f+1$ chunks are stored on correct nodes, which are sufficient to reconstruct the original block. In this way the storage consumption per block is reduced from $O(n)$ to $O(1)$. However, this method is still encountered with the following challenges.

1) *Availability of block data.* How to guarantee all blocks never become lost in Byzantine environment. When $2f+1$ sub-blocks are encoded with f parities in PBFT, it is impossible to ensure all $2f+1$ non-faulty nodes have reserved the correct chunks. Because the agreement of a block is reached for a node when it receives $2f+1$ commit votes from others, it only promises that at least $f+1$ non-faulty nodes will commit the block due to the existence of f faulty nodes.
2) *Scalability of system adjustment.* How to support flexible scalability, new nodes added to blockchain. The arrival of a new node means redistribution of current data, i.e, historical blocks need to be re-encoded among all nodes. However, since no node preserves entire blockchain, it is challenging to ensure that all correct nodes have accomplished re-encoding in distributed system with Byzantine fault.
3) *Efficiency of data access.* How to enhance read performance on encoded partitions distributed over nodes. Compared with full-replication manner, the read performance of blockchain declines due to data exchange among nodes for decoding. Specifically, to recover a block, a node needs to request a set of chunks from sufficient nodes, which incurs $O(n)$ network transmissions with n participants.

In view of above challenges, we design a Byzantine fault-tolerant storage engine for permissioned blockchain, which breaks the limitation of full-replication manner. In addition to the three challenges, how to check the data received for a node is also an important problem. In this study, we encode blocks by suitable RS coding according to f, which ensures there are sufficient chunks distributed over non-faulty nodes for decoding. The major contributions are summarized below.

- Design a Byzantine fault-tolerant storage engine called *BFT-Store* via EC for permissioned blockchain system, which reduces the storage complexity per block from $O(n)$ to $O(1)$ and improves the overall storage capability for the first time.
- Propose two key designs to achieve scalability and speed up read performance. First, we propose a four-phase re-encoding protocol based on PBFT, which promises the availability of all blocks. Second, we adopt a cache structure and multiple replication manner to ensure access efficiency of blocks.
- Give a formal proof of the correctness for BFT-Store on two facts: i) all blocks never become lost while the assumption for BFT holds; ii) the storage capability of BFT-Store grows with the number of nodes.
- Integrate BFT-Store into an open-source blockchain system Tendermint [11], and the extensive experiments confirm the scalability, availability and efficiency of BFT-Store compared with full-replication manner.

The remainder of this paper is structured as follows. Section 2 reviews related work and provides necessary background. Section 3 explains the problem settings. Section 4 gives the design of BFT-Store. Section 5 presents an online re-encoding protocol. Section 6 proves the correctness of our design. Section 7 contains a thorough set of experiments and Section 8 concludes the paper.

2 Related Work

BFT Protocol. Recently, blockchain has attracted a lot of attentions from industry and acadamic fields, including system protocols, consensus protocols [12], security, storage [13], [14], [15], and benchmarking [16]. BFT protocols have been extensively studied in the traditional distributed systems. As a piece of early influential work, the DLS [17] protocol achieves safety and liveness at the expense of $O(n^4)$ communication cost, which is prohibitively high for a large cluster. A decade later, Castro and Liskov propose PBFT [2], which incurs $O(n^2)$ transmissions where there are $n \geq 3f+1$ nodes, among which f are malicious. PBFT is often adopted to obtain consensus on a block in permissioned blockchain system, such as Tendermint [11] and Hyperledger Fabric 0.6 [18]. Meanwhile, in many protocols, the same node stays as the leader unless a view change occurs.

In classic PBFT protocol, each node has a private and public key pair and knows all others' public keys in advance. PBFT is a three-phases protocol including *Pre-Prepare*, *Prepare* and *Commit*. In an ordinary case, the consensus will be reached by all correct nodes with two rounds of voting *Prepare* and *Commit*. In *Pre-Prepare* phase, the leader proposes a block to all nodes. Subsequently, in *Prepare* phase, each node votes for the proposed block if it is correct and completed by broadcasting signed vote message. Upon receiving $n-f$ votes with correct signature for the proprosed block, each node broadcasts its vote for the block again in *Commit* phase. Finally, a node commits the block once receiving more than $n-f$ votes again.

Replication and Erasure Coding. As a standard approach for fault tolerance, replication can be categorized into two kinds, including synchronous [19] and asynchronous [20]. In recent years, erasure coding has gained favor and increasing adoption as an alternative to data replication because they incur significantly less storage overhead, while maintaining equal (or better) reliability. Many works optimize the coding efficiency and recovery bandwidth, like local reconstruction codes [21], piggyback codes [22] and lazy recovery [23].

Reed-Solomon coding [10] the most widely used EC, given a set of K data blocks, encodes them into $N=K+M$ chunks with M parities, denoted by (K,M)-RS. Both the original block and parity are termed as chunk in this paper. The coding is done such that any K out of N chunks are sufficient to reconstruct the original data, which can tolerate the absence of M chunks. For example, in (4,2)-RS coding, 4 MB of user data is divided into four 1 MB blocks. Then, two additional 1 MB parities are created to provide redundancy. RS coding computes parities according to its data over a finite field by the following formula:

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \alpha_1^1 & \cdots & \alpha_1^{K-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_{M-1}^1 & \cdots & \alpha_{M-1}^{K-1} \end{bmatrix} \times \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_K \end{bmatrix} = \begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_M \end{bmatrix}. \tag{1}$$

In Equation (1), $D_1, D_2, \ldots, D_K$ are K original blocks and $P_1, P_2, \ldots, P_M$ are M parities. Note that blocks $D_1, D_2, \ldots, D_K$ have identical size for encoding. The numbers $1, \alpha_1, \ldots, \alpha_{M-1}$ in the *Vandermonde matrix* are M different number. Given any K out of $K+M$ chunks, the original data $(D_1, D_2, \ldots, D_K)$ can be recovered. In this study, we combine BFT consensus protocol with erasure coding to circumvent the limitation of full-replication manner in permissioned blockchain system.

3 Problem Settings

3.1 Assumptions and Goals

As a Byzantine fault-tolerant storage engine which breaks the limitation of full-replication manner in blockchain system by combining BFT consensus protocol and erasure coding, BFT-Store still inherits the assumptions of BFT protocol. In short, there are two basic assumptions of BFT-Store presented as follows.

1) Out of the n nodes, at most $f=\lfloor\frac{n-1}{r}\rfloor$ $(r\geq 3)$ are faulty, which can fail in arbitrary way including explicitly malicious behaviour, and the remaining participants are correct, who strictly follow the protocol. The fault factor r determines the rate of faulty nodes, i.e, no more than $1/r$ incorrect nodes exist in system.
2) BFT-Store adopts *partial synchrony model* [17], using a known bound Δ and an unknown Global Stabilization Time (GST). After GST, all transmissions between two correct nodes arrive within time Δ, making the network synchronized most of the time.

The fault factor r is configurable in BFT-Store for various applications and a greater fault factor r means fewer faulty nodes in system. For example, fault factor r in classic PBFT protocol, $r=3$. In Zilliqa [24], the faulty nodes can be up to $1/4$ of all $(n\geq 4f+1)$, while at most $1/5$ nodes can be malicious $(n\geq 5f+1)$ in Ripple [25]. According to [26], given f Byzantine faulty node, no solution can deal with a system with fewer than $3f+1$ nodes. The BFT protocol requires the honesty of each node is static, which means an non-faulty node never becomes malicious and violates protocol. Note that an honest node can become crashed (faulty) due to any reason such as hardware problems, but cannot do evil, i.e., invoking conflicting messages. Otherwise, the safety of protocol cannot be promised according to the proof of [2]. The f faulty nodes count both Byzantine and non-Byzantine failures. Besides, different from public blockchain, nodes in a permissioned blockchain are usually equipped with high bandwidth, which guarantees the network is synchronized most of the time.

First of all, BFT-Store promises that the correct nodes can obtain any block through direct access or erasure decoding, which is critical to blockchain system. Then we consider to optimize the read performance of BFT-Store over partitioned chunks. Although the partitioning manner based on EC improves the storage efficiency, it reduces the read performance of system for remote data access. Furthermore, the performance will deteriorate if it is necessary to pull sufficient chunks from others to recover a block by decoding. In view of this, we design BFT-Store with the following goals:

1) *Availability:* Each block must be available for all correct nodes when the network is synchronized, which means no block will be lost if no more than $f=\lfloor\frac{n-1}{r}\rfloor$ faulty nodes exist.
2) *Scalability:* As the number of all nodes in BFT-Store grows, the storage complexity per block keep constant while the entire storage capability of system is improved.
3) *Efficiency:* With satisfaction of goals availability and scalability, compared with full-replication manner, the loss of read performance of BFT-Store should be as small as possible.

3.2 Concepts and Notations

Instead of encoding each block, BFT-Store encodes every $n-2f$ original blocks into n chunks with $2f$ parities via a $(n-2f,2f)$-RS, and each node preserves a unique chunk. According to the assumption of BFT, at least $n-2f$ chunks are stored on non-faulty nodes, and all blocks can be recovered based on these chunks by RS decoding. Specifically, a node just needs to request a set of chunks from $n-2f$ nodes to recover a original block by decoding when it fails to pull the block from others directly. Note that BFT-Store employs a $(n-2f,2f)$-RS rather than $(n-f,f)$-RS, which is

TABLE 1
The Important Symbols Used in This Article

Symbol	Description	Symbol	Description
r	fault factor	$\mathcal{R}_e^{(s)}$	eth coding round
$N_t(B)$	target node(s)	s	schema version #
B_i^e	original block	$\mathcal{C}_e(i)$	chunk set of N_i
C_i^e	chunk	τ	permutation func
P_i^e	parity	$P_r(B)$	reachable PR
$P_{dr}(B)$	direct reachable PR	c	# of replicas

discussed in Section 6. Before deep discussion, we give some basic concepts and notations in the following:

1) *Schema and version number s:* For a (K, M)-RS, (K, M) is called as the schema of RS coding. Whenever K or M changes, a new RS schema is applied and assigned a version number s to reflect the version evolution of schema.
2) *Coding Round $\mathcal{R}_e^{(s)}$:* The process of encoding $(r-2)f+1$ original blocks by RS coding is considered as a coding round. Each coding round has a unique sequence number e which starts at 0. The eth coding round with schema version number s is notated with $\mathcal{R}_e^{(s)}$.
3) *Block, parity and chunk:* In coding round $\mathcal{R}_e^{(s)}$ with (K, M)-RS, the $K=(r-2)f+1$ original blocks are denoted by $\{B_0^e, \ldots, B_{K-1}^e\}$, and the $M=2f$ parities are $\{P_0^e, \ldots, P_{M-1}^e\}$. Both blocks and parities are called chunks, represented as $\{C_0^e, \ldots, C_{N-1}^e\}$ $(N=n)$.
4) *Target node $N_t(B)$:* The target node $N_t(B)$ for a block B refers to node storing B locally.
5) *Reachable and direct reachable*: Block B is *reachable* for a node if B can be accessed without decoding, and is *direct reachable* if B is stored directly.

Reachability is more strict than availability. A block is reachable when its target node is correct. Let $P_r(B)$ and $P_{dr}(B)$ denote the probability that block B is reachable or direct reachable respectively. Table 1 lists important symbols used throughout this study.

4 Design

4.1 Architecture

Fig. 1 presents the architecture of BFT-Store, a storage engine for blockchain with Byzantine fault tolerance. Each user (termed as client) sends transactions to blockchain system. Transactions are packaged as a block for consensus among nodes and are then written into BFT-Store for persistence. Client reads blocks by sending requests to a node in BFT-Store. BFT-Store has four main components: *RS engine*, *read engine*, *recovery engine* and *scale-out engine*.

RS engine contains two sub-modules, encoder and decoder. Encoder buffers the blocks received from consensus, and encodes them into chunks. Each node stores a fragment of all chunks and discards the rest chunks. Note that there is no centralized node in BFT-Store, which encodes blocks into chunks and dispatches them to all nodes. Instead, each node needs to encode blocks independently without any information exchange. Once all correct nodes follow the same rule for encoding, the availability of each block is promised in BFT-Store. Decoder pulls chunks from others through underlying network and recovers blocks by RS decoding.

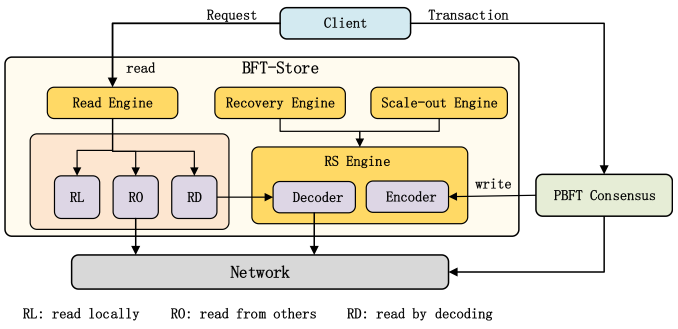


Fig. 1. Architecture of BFT-Store on each node.

Read engine receives read requests from clients and responds with the target blocks. There are three different cases when read engine handles a request to access block B. If block B is stored locally, read engine sends it back to client without any network communication. If the target node $N_t(B)$ is correct, read engine can pull it from $N_t(B)$. However, if the target node $N_t(B)$ is malicious and keeps silent, read engine has to recover B through decoder in RS engine. Particularly, read engine verifies all data from others to avoid recovering incorrect block data by decoding.

Recovery engine is used by a crashed node to recover block data. An honest node may crash for many reasons, but blocks are generated by the consensus of blockchain continuously. Hence, the crashed node will take the initiative to recover block data once it is restarted. To acquire blocks, recovery engine recovers original blocks through decoder of RS engine round by round. Then crashed node encodes them through encoder. However, a problem arises that the crashed node cannot to check the correctness of chunks received from others. We give a solution of this problem in Section 4.3.

Scale-out engine starts to work once new nodes join in blockchain system. Upon such a situation, the RS schema changes and historical blocks must be re-encoded correspondingly. To re-encode blocks, each node has to exchange data with others since no node stores the entire blockchain. BFT-Store employs a leader-driven re-encoding protocol, in which leader is in charge of recovering original blocks and broadcasting them to all nodes for re-encoding. In addition, before deleting old chunks, each node ensures that the rest have accomplished re-encoding to guarantee the availability of blocks. See Section 4.3 for details.

4.2 Improvement for Read Performance

Although the partition for block data based on EC improves space efficiency significantly, the read performance of system decreases for decoding and network transmissions. Particularly, there are two main reasons for performance degradation.

1) If target node $N_t(B)$ is malicious and keeps silent, block B is not reachable and can only be recovered by decoding, which entails $O(n)$ network transmissions. The probability of this event in a node is $1-P_r(B)=f/n \approx 1/r$, which is a high rate.

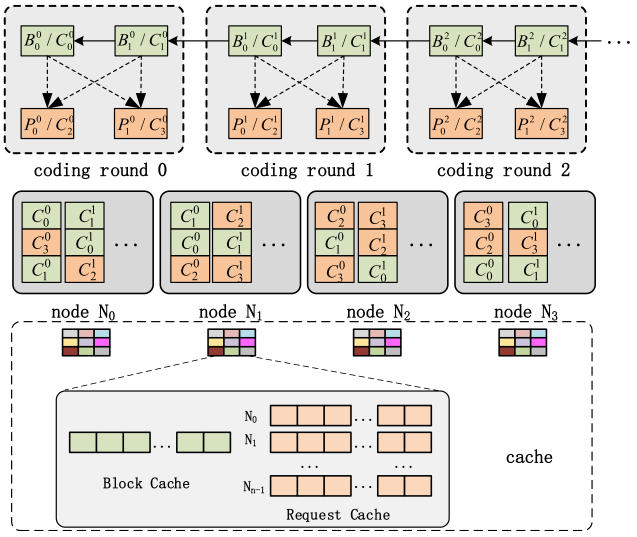


Fig. 2. Example of BFT-Store with (2,2)-RS coding and 3 replicas for each chunk.

2) The probability $P_{dr}(B)$ for a block B is $1/n$ close to 0 with a large number n, which means it is necessary for a node to obtain the block data via network transmission with high confidence, $1 - P_{dr}(B) \approx 1$.

To alleviate the degradation of read performance, we employ replicating and caching techniques to raise $P_r(B)$ and $P_{dr}(B)$ to improve read performance.

Replicating. In coding round $\mathcal{R}_e^{(s)}$, each chunk C_i^e is replicated over c different nodes, where c is the number of replicas. Hence, a block has c different target nodes and notation N_t refers to a set of nodes rather than single node. Then block B is not reachable only if all of its target nodes $N_t(B)$ are faulty, thereby the probability that B is not reachable is calculated as follow.

$$1 - P_r(B) = \prod_{i=0}^{c-1} \frac{f-i}{n-i} \le \prod_{i=0}^{c-1} \frac{f}{n} = \left(\frac{f}{n}\right)^c \le r^{-c}.$$

The probability $1 - P_r(B)$ is less than r^{-c} which decreases exponentially with the growth of c, and the probability $P_r(B)$ is greater than $1 - r^{-c}$. For example, when $c = 3$ and $r = 3$, $P_r(B)$ is greater than $1 - 1/27 = 26/27 \approx 96\%$; and $P_r(B)$ exceeds 99 percent when $c = 5$ and $r = 3$. Suppose the average size of a block is T, then the storage consumption for every $n - 2f$ blocks is cnT. Thus the average storage overhead per block is $\frac{cnT}{n-2f} \approx \frac{crT}{r-2}$, which is a constant complexity $O(1)$.

Example 1. The upper part of Fig. 2 illustrates an instance of BFT-Store with schema (2,2)-RS versioned by 0. In a coding round, each chunk is replicated over 3 different nodes and each node preserves three different chunks. For example, in coding round $\mathcal{R}_e^{(0)}$, blocks B_0^0 and B_1^0 are encoded into 4 chunks, including two block chunks C_0^0, C_1^0 and two parity chunks C_2^0 and C_3^0. Meanwhile, node N_0 stores three chunks, C_0^0, C_3^0 and C_1^0, and chunk C_0^0 is replicated on three target nodes N_0, N_1 and N_3 separately.

With c replicas for each chunk, each node N_i has to store c different chunks, namely chunk set $\mathcal{C}_e(i)$, in coding round $\mathcal{R}_e^{(s)}$. In BFT-Store, each node N_i computes chunk set $\mathcal{C}_e(i)$

-	$\mathcal{C}_e(0)$	$\mathcal{C}_e(1)$	$\mathcal{C}_e(2)$	$\mathcal{C}_e(3)$
τ_1	0	1	2	3
τ_2	1	2	3	0
τ_3	3	0	1	2

independently without any information exchange. In order to achieve this, all nodes are sorted by their public keys and the position in this list is treated as the global index for each node. In this way, the sorted node list is identical among all honest nodes because each node knows all others' public keys. To construct $\mathcal{C}_e(i)$, node N_i with index i uses hash value of coding round number and sequence number of RS schema $Hash(e||s)$ as seed to generate c different permutations $\{\tau_1, \ldots, \tau_c\}$ of $\{0, \ldots, n-1\}$. Then, the chunk set of node N_i in coding round $\mathcal{R}_e^{(s)}$ is $\mathcal{C}_e(i) = \{C_{\tau_1(i)}^e, C_{\tau_2(i)}^e, \ldots, C_{\tau_c(i)}^e\}$. Since the permutations are generated based on e and s, all honest nodes must get the same permutations. Therefore, the chunk set of different honest nodes must not be identical. This ensures at least $n - 2f$ different chunks are available, which are sufficient to recover all original blocks. Furthermore, each node needs not to maintain an index structure to indicate the target nodes of each chunk, since it can recompute the permutations based on s and e. Then, the target nodes of a chunk C_i^e are $\{N_{\tau_1^{-1}(i)}, \ldots, N_{\tau_1^{-1}(i)}\}$, where τ_i^{-1} is the inverse of τ_i which can be calculated according to τ_i. Note the multiple replicas of each chunk are used to improve the reading performance rather than promise the availability of block. Even when $c = 1$, all blocks still keep recoverable via RS decoding based on chunks of non-faulty nodes.

However, if a node may store two identical chunks in a coding round, the decline of confidence $P_r(B)$ for block B may lower the read performance as well. To avoid this problem, node N_i employs the hash values of blocks to generate c different numbers ranging from 0 to $n-1$, represented by $A = \{a_1, \ldots, a_c\}$. Then the chunk set $\mathcal{C}_e(i)$ is computed as follow: $A + i = \{\tau_j(i) = a_j + i \bmod n : 1 \le j \le c\}$ $(0 \le i \le n-1)$. For the sake of simplicity, we term number k as chunk C_k^e under the condition without confusion. Example 2 illustrates an instance that each node stores 3 different chunks in a coding round when $n = 4$ and $c = 3$.

Example 2. When $n = 3 \times 1 + 1$ and $c = 3$, we generate 3 different permutations, such as $\tau_1 = (0, 1, 2, 3), \tau_2 = (1, 2, 0, 3), \tau_3 = (3, 0, 1, 2)$. The chunk set of N_1 is $\mathcal{C}_e(1) = \{\tau_1(1) = 1, \tau_2(1) = 2, \tau_3(1) = 0\}$, while N_3 stores chunk set $\mathcal{C}_e(3) = \{3, 3, 2\}$. The chunk C_3^e appears in chunk set $\mathcal{C}_e(3)$ twice, which declines the read performance. To promise no duplicated chunk, we pick $A = \{0, 1, 3\}$. Then four chunk sets are: $\mathcal{C}_e(0) = A + 0 = \{0, 1, 3\}$, $\mathcal{C}_e(1) = A + 1 = \{1, 2, 0\}$, $\mathcal{C}_e(2) = A + 2 = \{2, 3, 1\}$ and $\mathcal{C}_e(3) = A + 3 = \{3, 0, 2\}$. The three permutations are shown as follows.

Lemma 1 shows the correctness of our design. In addition, for any two nodes, we hope there are fewer common chunks in their chunk sets, which makes chunks are distributed more evenly over all nodes. Ideally, any two nodes have at most one common chunk in a coding round. If so, a node can receive $n - 2f$ different chunks from fewer nodes earlier while recovers blocks by decoding. This is solved in

Appendix A.1, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2020.3012668.

Lemma 1. *For node N_i, there is no duplicated chunk in chunk set $\mathcal{C}_e(i)$ based on the proposed solution.*

Proof. By the way of contradiction, assume node N_i stores two same chunks $C^e_{\tau_j(i)}$ and $C^e_{\tau_k(i)}$ ($j \neq k$) at the same time in coding round $\mathcal{R}_e^{(s)}$.

$$\tau_j(i) = \tau_k(i) \Leftrightarrow (a_j + i) \equiv (a_k + i) \bmod n \Leftrightarrow a_j \equiv a_k \bmod n.$$

Since the elements of A are different, j must be equal to k, which is contradicted against the setting of A. Consequently, the lemma holds. □

Caching. The probability $P_{dr}(B)$ that B is direct reachable is quite low even each node stores c chunks locally in a coding round. A straightforward solution is cache technology. In existing public blockchain systems, such as Ethereum [27], it only concerns on the recent several blocks to check the validity of current world state. Hence, the latest blocks can be seen as hot data and the old blocks can be seen as cold data. Each node sets a buffer to cache a batch of blocks generated recently without partition. The requests for recent blocks can be responded by fetching them from the buffer without any network transmission. In addition to the buffer, BFT-Store adopts two kind of caches to improve the read performance: *block cache* and *request cache*.

Block cache is used to buffer the blocks requested frequently which are stored on other nodes. A typical cache is *Least Recently Used* (LRU) cache. When receiving target block from others, each node updates block cache by it. A node also sets a request cache for every node to record the block which is recovered through decoding on the node. If node N_i wants to recover a block B^e_k by decoding, it has to obtain sufficient different chunks of coding round $\mathcal{R}^s_e$ from others. Then other nodes record (e, k, s) in request cache for N_i, indicating that it has ever requested block B^e_k. At this time, N_i can be seen as a target node for B^e_i because it may cache the block in its block cache. This is helpful to increase probability $P_r(B)$ for block. Each node sets a block cache and $n-1$ request caches for other nodes. Block cache buffers blocks from others and request cache just records index information of requested block. On the lower part of Fig. 2, it presents the structure of cache in BFT-Store. Each node sets a block cache to buffer blocks and $n-1$ request caches for other nodes.

4.3 Encoding and Reading

We detail the process of encoding and reading of BFT-Store in this section. To recover a block by decoding, a node needs to pulls sufficient chunks from others. Thereby, a critical problem must be solved, where a node cannot verify the completeness and correctness of the received chunks. A naive approach is that each node preserves the hash values of all chunks in a coding round, and then uses them to check received chunks. However, this will increase the storage consumption per block to $O(n)$, because all nodes store the hash value of every chunk. Furthermore, this approach cannot deal with data recovery of crashed node. Similar to that in traditional distributed systems, honest node N_i may become crashed for many reasons. In contrary to Byzantine node, honest node will try to recover data when it is restarted, which requires to synchronize chunks from others to participate in consensus of system again. Unfortunately, without the hash values of the newly generated chunks on the crashed node due to it's absence in consensus during crash, it is impossible for this node to check the synchronized chunks.

In BFT-Store, we adopt *threshold signature* [28] to provide verification for each chunk. An (n, t)-TS threshold signature on a message m is a single, constant-sized aggregate signature that passes verification if and only if at least t out of the n participants sign m. Note that the verifier does not need to know the identities of the t signers. In BFT-Store, an $(n, n-f)$-TS is deployed among all nodes. When accomplishes encoding for a coding round $\mathcal{R}_e^{(s)}$, each node piggybacks its signatures of all chunks of $\mathcal{R}_e^{(s)}$ to the vote message in consensus for the first block in next coding round $\mathcal{R}_{e+1}^{(s)}$. When receives $n-f$ vote messages, each node N_i aggregates $n-f$ signatures, for each chunk in $\mathcal{C}_e(i)$ itself into a single signature, which is the proof of chunk, and stores the signature together with chunk. A node checks the correctness of a chunk received from other node by verifying the threshold signature of it. By this method, each node just needs to preserve a extra threshold signature for each chunk stored by itself for verification instead of hash values of all chunks.

Algorithm 1. Block Encoding of BFT-Store

```
Input: node $N_i$, coding round $\mathcal{R}_e^{(s)}$, original block set
       $\mathcal{B} = \{B^e_0, \ldots, B^e_{K-1}\}$
1:  $\mathcal{C} = \{C^e_0, \ldots, C^e_{n-1}\} \leftarrow$ RSEncode($\mathcal{B}$)
2:  $\mathcal{C}_e(i) \leftarrow$ CreateChunkSet($\mathcal{C}, i$)
3:  $\mathcal{H} \leftarrow \emptyset$ /*signature set of all chunks in $\mathcal{R}_e^{(s)}$*/
4:  for chunk $C^e_j$ in $\mathcal{C}$ do
5:    $\mathcal{H} \leftarrow \mathcal{H} \cup Sign(C^e_j)$
6:  broadcast $\mathcal{H}$ along with commit-vote message for block
    $B_0^{e+1}$ in coding round $\mathcal{R}_{e+1}^{(s)}$
7:  upon receive $n-f$ commit-vote messages $\mathcal{V}$ for $B_0^{e+1}$ do
8:    $\mathcal{H}_1, \ldots, \mathcal{H}_{n-f} \leftarrow n-f$ signature sets of nodes in $\mathcal{V}$
9:    $TS_e(i) \leftarrow \emptyset$
10:   for $j$ from $1$ to $c$ do
11:     /*create threshold signature for chunk $C^e_{\tau_j(i)}$*/
12:     $signs \leftarrow \{\mathcal{H}_1[\tau_j(i)], \ldots, \mathcal{H}_{n-f}[\tau_j(i)]\}$
13:     $ts \leftarrow$ Aggregate($signs$)
14:     $TS_e(i) \leftarrow TS_e(i) \cup ts$
15:   store $\mathcal{C}_e(i)$ and $TS_e(i)$
```

Algorithm 1 shows the pseudo-code of encoding for blocks on node N_i. First, $K = n - 2f$ original blocks of coding round e are encoded into n chunks $\mathcal{C} = \{C^e_0, \ldots, C^e_{n-1}\}$ by function $RSEncode()$ (line 1). Note that the block smaller than the maximal one is appended with '0' for alignment. Second, function $CreateChunkSet()$ computes the chunk set $\mathcal{C}_e(i)$ of N_i based on chunks $\mathcal{C}$ (line 2). After that, node N_i signs all chunks in $\mathcal{C}$ and gets the signature set $\mathcal{H}$ (lines 3-5). Third, N_i broadcasts signature set H along with commit-vote message for block B_0^{e+1} of next coding round $\mathcal{R}_{e+1}^{(s)}$ (line 6). This phase is embedded in the progress of PBFT consensus without additional round of network communication.

Forth, upon receiving $n-f$ commit-vote messages for block B_0^{e+1}, each node extracts the signature sets of different nodes $\mathcal{H}_1,\ldots,\mathcal{H}_{n-f}$ from commit-vote messages $\mathcal{V}$ (line 8). Then, it computes the threshold signatures for all chunks in its chunk set $\mathcal{C}_e(i)$ (lines 9-14). Function $Aggreage()$ aggregates $n-f$ different signatures into single constant-size signature which can be verified by the threshold public key (line 12). Finally, node N_i stores chunk set $\mathcal{C}_e(i)$ and its corresponding threshold signatures $TS_e(i)$ in coding round $\mathcal{R}_e^{(s)}$.

Algorithm 2. Block Reading of BFT-Store

```
 Input: block height $h$ of target block
 Output: target block $B_t$
 1: (Processing on node $N_c$)
 2: if block with height $h$ stored locally then
 3:     $B_t \leftarrow$ block with height $h$
 4:     return $B_t$
 5: $N_t(B_t) \leftarrow$ GetTargetNodes($h$)
 6: $msg \leftarrow \langle \mathrm{BLOCK}, h, e, s\rangle_{\sigma_c}$
 7: SendMessages($N_t(B_t), msg$)
 8: upon receive block $B_t$ from $N_t(B_t)$ do
 9:     UpdateBlockCache($B_t$)
10:     return $B_t$
11: upon timeout for reception from $N_t(B_t)$ do
12:     $nnodes \leftarrow$ RandomNodes($n-f$)
13:     $msg \leftarrow \langle \mathrm{DECODE}, h, e, s\rangle_{\sigma_c}$
14:     SendMessages($nnodes, msg$)
15: upon receive $n-2f$ different chunks $CS$ from $nnodes$ do
16:     $\mathcal{B} \leftarrow$ RSDecode($n-2f, 2f, CS$)
17:     $B_t \leftarrow \mathcal{B}$.get($h$), UpdateBlockCache($B_t$)
18:     return $B_t$
19: (Processing on node $N_i$)
20: upon receive message $\langle \mathrm{BLOCK}, h, e, s\rangle_{\sigma_c}$ do
21:     if block with height $h$ stored locally then
22:         $B_t \leftarrow$ block with height $h$
23:         SendChunks($\{B_t\}, N_c$)
24: upon receive message $\langle \mathrm{DECODE}, h, e, s\rangle_{\sigma_c}$ do
25:     UpdateRequestCache($h, N_c$)
26:     if block with height $h$ stored locally then
27:         $B_t \leftarrow$ block with height $h$
28:         SendChunks($\{B_t\}, N_c$)
29:     else
30:         $\mathcal{C}_e(i) \leftarrow$ chunks of round $e$ stored by $N_i$
31:         SendChunks($\mathcal{C}_e(i), N_c$)
```

Algorithm 2 presents how to process a read request for target block B_t on node N_c and others.[2] First, when node N_c receives a read request for B_t with height h, it checks whether B_t is stored in local cache or its chunk set. If so, it returns the target block B_t directly (lines 2-4). Otherwise, node N_c sends a block message $< \mathrm{B}LOCK, h, e, s >_{\sigma_c}$ to target nodes computed by function $GetTargetNodes()$, including nodes in request cache, to fetch B_t (lines 5-7), where σ_c is the signature of node N_c for security. Second, when a node N_i receives block message from N_c for block with height h, it sends back target block B_t to N_c if stores B_t locally (lines 20-23). Node N_c updates block cache for B_t by function $UpdateBlockCache()$ and returns when receives it from target nodes (lines 8-10). Third, when the timer for message reception from target nodes expires i.e, the target node is not reachable, N_c broadcasts a decode message $< \mathrm{DECODE}, h, e, s >_{\sigma_c}$ to $n-f$ nodes randomly to pull sufficient chunks for decoding (lines 11-14). Note that the process for the case that the target node is malicious and sends back incorrect block is not presented in Algorithm 2, because node N_c can detect this misbehavior by verifying the threshold signature and then tries to recover block by decoding again, which is same as the codes in lines 11-4. There are at least $(n-f)-f=n-2f$ correct nodes among any $n-f$ nodes, which reaches the quorum to perform decoding operation. Forth, upon receiving decode message from N_c, node N_i updates request cache for node N_c by function $UpdateRequestCache()$ (line 25). If the target block is cached by N_i, it sends B_t back to N_c directly (lines 26-28). Otherwise, N_i sends its corresponding chunk set $\mathcal{C}_e(i) = \{C^e_{\tau_1(i)},\ldots,C^e_{\tau_c(i)}\}$ to N_c of coding round $\mathcal{R}_e^{(s)}$ (lines 29-31). Last, node N_c recovers all original blocks of coding round $\mathcal{R}_e^{(s)}$ by decoding when it receives $n-2f$ different chunks (lines 15-16). Then, node N_c updates its block cache and returns target block B_t (lines 17-18). Note that node N_c also checks the correctness of received chunks by verifying the threshold signatures of them. Since the original blocks are meaningful to users, the node, storing original block, will gain more workload for read in each coding round. To balance the workload, BFT-Store assigns chunks to all nodes randomly in each coding round by selecting a new seed. A node storing parity in previous round may store block in next round.

5 Online Re-Encoding

5.1 Schema Update for Network Change

It is crucial to consider the addition of new node or removal of existing node in permissioned blockchain. According to the assumption $n \geq rf+1$, n always can be represented in the form $n=rf+i$ $(1 \leq i \leq r)$, and the schema of RS coding is $((r-2)f+i, 2f) = (n-2f, 2f)$. If $n = rf+r+1$, n can be represented as $n = rf'+1$ where $f'=f+1$ represents the number of faulty nodes. With the addition or deletion of nodes, the schema of RS coding needs to be updated since f or i may vary. There are two kinds of update for RS schema: i) just the number i varies; ii) the number of faulty nodes f changes. Suppose the number of nodes in systems change from n to n' such as $n'=n+k$, where a positive integer k represents the addition of new node(s) and a negative integer k means deletion of existing node(s). Then the threshold of faulty nodes becomes $\lfloor \frac{n'-1}{r} \rfloor$, and the new schema of RS is $(n'-2f, 2f')$ correspondingly. As the schema of RS changes, all historical blocks need to be re-encoded and redistributed over all nodes. In BFT-Store, we hope the re-encoding of old blocks does not affect the consensus of new blocks and data access service.

Example 1. Fig. 3 shows an example of schema update for addition of a new node when $r=3$, $f=2$ and $n=3f+1=7$. When $n=3f+1=7$, the schema (3,4) is applied, and every 3 blocks are encoded with 4 parities, as shown in real line boxes in the middle of Fig. 3. With a new node, the parameter n changes to $3f+2$ and the schema transfers from (3,4) to (4,4). Thus every 4 blocks

2. For brevity, the verification and check of messages, containing signatures and state information, are omitted.

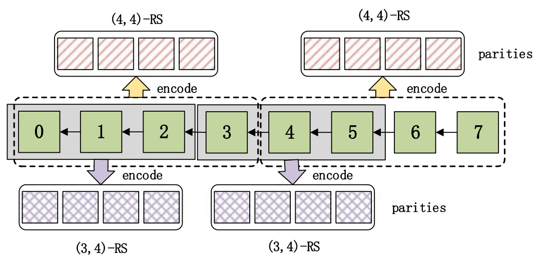


Fig. 3. Example of schema update.

are encoded with 4 parities as shown in the dotted line boxes of Fig. 3. In this case, parameter i changes from 3 to 4 while f remains unchanged. However, when $n = 9$ and a new node joins, then the parameter f changes to 3 and the schema of RS coding will transfer from (5,4) to (4,6). Here, parameters i and f change at the same time. In the later case, the storage capability of BFT-Store will decline because more parities are created with the growth of f. See Section 6 for details. On the contrary, the schema will change with the removal of existing nodes. For example, the schema transfers from (4,4) to (3,4) when a node out of 8 participants are removed.

5.2 Re-encoding for Historical Blocks

In this section, we focus on the addition of new node, and the removal of node can be dealt with in similar way. To re-encode historical blocks, a naive method is that each node recovers the whole blockchain by reading partitions from other nodes, and then re-encodes these blocks with new RS schema. However, it will involve huge network transmissions due to chunk immigration which is an all-2-all communication. In addition, it is difficult for a node to judge whether the progresses of re-encoding on others are accomplished. This may result in unexpected loss of blocks during re-encoding. For instance, node N_i deletes block B when it finishes re-encoding, but other nodes have not synchronized B from N_i. Sequentially, block B becomes unavailable and cannot be recovered forever. BFT-Store deploys a leader driven re-encoding, and assumes that new node is added into system at a low frequency. Commonly, in permissioned blockchain, nodes to participate are those that are granted permission only [29], which is against the free addition or removal of public blockchain, thus it is reasonable to assume the frequency of network change is low. Different from consensus protocol PoW [1], [3] deployed in public setting which can support a large number of nodes, the performance of PBFT becomes worse significantly as the network grows [16]. Thus, the BFT-based blockchain (e.g., permissioned blockchain in this work) support fewer participants to keep excellent performance.

Before re-encoding, the new node broadcasts a special transaction TX_{new} to the network. When the block containing TX_{new} is committed by node N_i, the new node is added to system in the view of node N_i. The PBFT consensus algorithm guarantees that all honest nodes commit TX_{new} at the same block height which can be seen as the logical timestamp of system. Then, nodes start to re-encode historical blocks. BFT-Store arranges a node as the leader L to coordinate the progress of re-encoding, and all nodes move

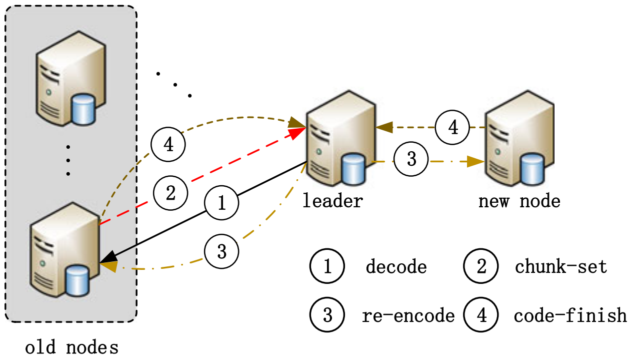


Fig. 4. Re-encoding for schema update.

through a succession of configuration called *windows*. In a window, a node is the leader and the others are *followers*. With the addition of new node, the number of all and faulty nodes become n' and f' respectively, and the version number of new schema is $s + 1$. Fig. 4 illustrates the progress of re-encoding in a window coordinated by leader L. The re-encoding of historical blocks is involved round by round, and there are four phases for re-encoding of a coding round $\mathcal{R}_e^{(s+1)}$ in a window. Each phase corresponds with a type of message transmitted between leader and followers, including decoding, chunk-set, re-encoding and code-finish. The four phases are detailed as follows.

1) In the first phase, leader broadcasts the decode message to all followers excluding the new nodes. If the blocks of coding round $\mathcal{R}_e^{(s+1)}$ are not available locally, L broadcasts a decode message to old nodes for chunks of coding round $\mathcal{R}_e^{(s+1)}$.
2) In the second phase, followers response with their chunk-set message. Each follower N_i responds to leader L with its chunk set for old schema s when receives decode message.
3) In the third phase, leader broadcasts re-encode message to all followers including the new nodes. Leader L recovers all original blocks $\mathcal{B}$ of coding round $\mathcal{R}_e^{(s+1)}$ once receives $n - 2f$ different chunks from followers. Then, it broadcasts a re-encode message $< \mathrm{RE\_ENCODE}, \mathcal{B}, \mathcal{V}, s+1, w, e >_{\sigma_L}$ to all followers, where $\mathcal{V}$ is a set of $n' - 2f'$ finish-code messages explained later.
4) In the last phase, followers send their coding-finish to the leader. When follower N_i receives a re-encode message from the leader, it re-encodes blocks $\mathcal{B}$ into n' chunks and stores its chunk set $\mathcal{C}_e(i)$ in coding round $\mathcal{R}_e^{(s+1)}$. Then, node N_i sends a code-finish message $< \mathrm{CODE\_FINISH}, e, s+1 >_{\sigma_i}$ back to leader L indicating its accomplishment of re-encoding for round $\mathcal{R}_e^{(s+1)}$.

Note that when the leader L receives $n' - f'$ code-finish messages for current coding round $\mathcal{R}_e^{(s+1)}$, it starts to re-encode blocks for the next coding round $\mathcal{R}_{e+1}^{(s+1)}$. To ensure that all followers are known that the majority of all correct nodes have accomplished re-encoding for round $\mathcal{R}_e^{(s+1)}$, leader L broadcasts received $n' - f'$ code-finish messages, along with the re-encoding message of the next round $\mathcal{R}_{e+1}^{(s+1)}$. We can still adopt *threshold signature* to aggregate

$n' - f'$ code-finish messages into a single message as explained in Section 4.3. A follower removes chunks with old schema when ensures that at least $n' - f'$ nodes have accomplished re-encoding by verifying signatures in $\mathcal{V}$. The coding round $\mathcal{R}_e^{(s+1)}$ becomes stable for a follower when it receives a $\mathcal{V}$ containing at least $n' - f'$ correct finish-code messages (or single message with threshold signature). By this method, BFT-Store promises that all blocks are available before re-encoding for coding round $\mathcal{R}_{e+1}^{(s+1)}$.

During re-encoding, the blockchain system still can provide data access service for any block and new blocks are produced continuously. The newly generated blocks are encoded with new RS schema among n' nodes and the old blocks that have not been re-encoded can be recovered by the old schema. Since all nodes are static, old blocks keep available which are stored over n nodes before re-encoding. To determine the target node(s) of each block when reading, each node recompute the permutation based on parameters e and s and then gains the target node(s). Therefore, even in the process of re-encoding, each block is available and BFT-Store provides service properly. It emphasizes that the newly added node can be honest or malicious. If the node is honest, it will follow the protocol and stores some chunks after re-encoding. If the node is malicious, it may discard all chunks. However, this misbehavior does not affect the progress of re-encoding on other nodes and the chunks stored on it can be recovered by decoding.

The process of removal of node can be finished in a similar way. In the second phase of re-encoding, the leader collects chunks from all nodes, and just broadcasts re-encode message to nodes except the one to be removed. The difference is that the node only can be removed from system after the accomplishment of re-encoding to guarantee the availability of blocks.

5.3 Leader Rotation

Actually, the leader may be malicious[3] or become crashed during the re-encoding in a window. Hence, leader rotation is inevitable. The leader rotation is triggered by timeouts that prevent followers from waiting for re-encode message carrying original blocks from the leader. A follower starts a timer when it sends code-finish message to leader. It stops the timer when receives re-encode message from the leader. If malicious leader keeps silent, the timer of follower N_i expires and N_i starts a leader rotation to move the system to windows $w + 1$. The steps of leader rotation are as follows.

1) When the timer of follower N_i expires, it stops accepting messages and sends a window-change message $< \mathrm{WINDOW\_CHANGE}, w+1, e, s+1 >_{\sigma_i}$ to the next leader where e is the largest coding round number that becomes stable for N_i. The leader of each window is elected in round robin or any other manner.
2) When the leader L receives $n' - f'$ valid window-change messages for window $w + 1$, it broadcasts a new-window message $< \mathrm{NEW\_WINDOW}, e', w+1, s+1, \mathcal{O} >_{\sigma_L}$ to all followers, where $\mathcal{O}$ is a set of the valid window-change messages and e' is a coding round number. Then it enters window $w + 1$. The determination of e' is discussed later.
3) A follower accepts a new-window message for window $w + 1$ if it is signed properly, if the window-change messages are valid for window $w + 1$, and then the follower enters window $w + 1$.

After the above three steps, the new leader L can broadcast re-encode message for each coding round after the latest stable coding round $\mathcal{R}_{e'}^{(s+1)}$. The coding round number e' is computed by leader L as follows: i) sort $n' - f'$ window-change messages by coding round number in an ascending order; ii) select coding round number of the $(n' - 2f')$th message as the value of e'. Due to the existence of at most f' faulty nodes, out of the $n' - f'$ received window-change messages, at least $n' - 2f'$ of them are sent by correct followers. Coding round $\mathcal{R}_{e'}^{(s+1)}$ must become stable on one correct node N_i. Coding round $\mathcal{R}_{e'}^{(s+1)}$ is stable on N_i only if it ensures more than $n' - 2f'$ non-faulty nodes have finished re-encoding for $\mathcal{R}_{e'}^{(s+1)}$. Hence, re-encoding for the coding rounds with number less than e' is finished by the majority of followers which guarantees the availability of blocks. By here, the progress of leader rotation is accomplished.

6 Discussion and Correctness

In this Section, we sketch the correctness of availability and scalability of BFT-Store. Recalling the aforementioned assumption in this study that at most f malicious nodes out of n nodes such that $n \geq rf + 1$, it is different from classic PBFT consensus protocol when $n > 3$. However, it is convenient to apply our assumption on PBFT with a slight modification. In classic PBFT, each node updates its state to enter prepared or committed state in a consensus round upon receiving more than $n - f = 2f + i \ (i = 1, 2, 3)$ votes. The modified PBFT protocol is that each node waits for $n - f = (r-1)f + i \ (1 \leq i \leq r)$ votes before entering next state. By this, the modified PBFT protocol still satisfies the safety and liveness which can be proved in the same way as classic PBFT protocol. In the remaining of this section, we briefly discuss the availability and scalability of BFT-Store.

Availability. Here we discuss the setting of parameters for RS schema. Lemma 2 and 3 explain the reasons that parameters K and M are set to $n - 2f$ and $2f$ respectively.

Lemma 2. *In a coding round, at least f parities are necessary to guarantee the availability of all blocks.*

Proof. With (K, M)-RS coding, it can tolerate the loss of at most M chunks. The K original blocks can be recovered by decoding based on any K chunks. If $M < f$ and $c = 1$, at most f of $K + M$ chunks may be stored on faulty nodes. Therefore, it just guarantees that $K + M - f < K$ chunks are available which does not reach the quorum of (K, M)-RS. Consequently, at least f parities are needed. □

Lemma 3. *To promise the availability of blocks, the maximal value of parameter K is $n - 2f$ in BFT environment.*

Proof. There are two main reasons for this setting. First, during re-encoding, it just guarantees a leader at least collects $n - f$ code-finish messages from all nodes since f faulty nodes may keep silent without any message. When

3. A malicious leader can keep silent or send incorrect blocks to other nodes. It can even send original blocks to part of all nodes.

receives $n-f$ code-finish messages, the leader has no way to check whether all of them actually have completed the process, because the f faulty nodes may deceive the leader with forged finish-code messages. The only thing that is sure is there are at least $(n-f)-f=n-2f$ correct nodes among these $n-f$ nodes. Therefore, the leader must be able to recover all original blocks based the chunks stored on the $n-2f$ non-faulty nodes. If $c=1$, at most $n-2f$ different chunks are available from the $n-2f$ correct nodes. If $K>n-2f$, it cannot promise the availability of all blocks during re-encoding in BFT-Store.

The second reason is related to PBFT consensus protocol. In our modified PBFT protocol, a node commits block while receiving $n-f$ *prepare-vote* messages for it, which prevent other honest nodes from committing any conflict block. If block B is committed at height h by node N_i, it just ensures that at least $n-2f$ correct nodes have committed block B' at height $h-1$ according to $n-f$ prepare-vote messages. In other words, the consensus can be reached by system while at most f correct nodes are absent from consensus for any reason and the f faulty nodes follow the PBFT protocol. Then, the f faulty nodes keep silent in process of reading and only $n-2f$ non-faulty nodes store chunks. Similarly, all blocks must be recovered based the chunks stored on the $n-2f$ correct nodes. □

With an $(n-2f,2f)$-RS coding, all blocks are available even during re-encoding when the network is synchronized. If the network becomes asynchronized, nodes may fail to pull data from others. However, it will re-try to recover block after waiting for some time. Since the network is synchronized most of the time, there is no significant impact on BFT-Store.

Scalability. We analyze the average storage consumption per block and the change of whole storage capability of BFT-Store with addition of new nodes. First, we prove the storage complexity per block keeps constant, which is computed as follow:

$$\begin{aligned} S_B &= \frac{rf+i}{(r-2)f+i}cT = \frac{(r-2)f+i+2f}{(r-2)f+i}cT \\ &= \left[1+\frac{2}{(r-2)+\frac{i}{f}}\right]cT \le \left(1+\frac{2}{r-2}\right)cT, \end{aligned}$$

where T is average size of a block. For every $n-2f$ original blocks, its storage consumption is $cnT=(rf+i)cT$, thereby storage consumption per block S_B is $cnT/(n-2f)=(rf+i)cT/[(r-2)f+i]$. Observe that S_B decreases with the growth of i or decline of f. However, there is an upper bound $\left(1+\frac{2}{r-2}\right)cT$ for S_B, which is a constant variable. Hence, the storage complexity per block is $O(1)$.

Second, we discuss the change of storage capability of BFT-Store when new nodes join BFT-Store. In order to facilitate the discussion, the maximal number of original blocks that can be stored by all nodes in BFT-Store is adopted to measure the storage capability of system, denoted by $S(f,i)$ $(n=rf+i, 1\le i\le r)$. Besides, assume that all nodes are equipped with same storage space notated by S_n, thereby the total storage space of all nodes is nS_n. Then, the storage capability $S(f,i)$ is computed as follow:

$$S(f,i)=\frac{nS_n}{S_B}=\frac{(rf+i)S_n}{\frac{rf+i}{(r-2)f+i}cT}=\frac{[(r-2)f+i]S_n}{cT}.$$

Note that the storage capability $S(f,i)$ increases with the growth of f or i. However, $S(f,i)$ does not always increase with the growth of $n=rf+i$, because when number n increases, number i may decrease correspondingly. With more nodes $n'=r(f+1)+j$ $(1\le j\le r)$, the storage capability of BFT-Store is improved when $S(f+1,j)-S(f,i)$ is greater than 0 which is presented as follow:

$$\begin{aligned} & S(f+1,j)-S(f,i) > 0 \\ \Leftrightarrow & \frac{(r-2+j-i)S_n}{cT} > 0 \Leftrightarrow r-2+j-i > 0 \\ \Leftrightarrow & r+j-i > 2 \Leftrightarrow r(f+1)+j-(rf+i) > 2 \\ \Leftrightarrow & n'-n > 2 \Leftrightarrow n'-n \ge 3. \end{aligned}$$

According to above derivation, if f remains unchanged, storage capability $S(f,i)$ is improved with more nodes. When f grows, $S(f,i)$ increases only if at least 3 new nodes join BFT-Store. For instance, when $r=3$, $f=2$ and $n=3\times 2+3=9$, $S(2,3)$ is greater than $S(3,1)$ since there is only one new node, while $S(3,3)$ exceeds $S(2,3)$ for addition of 3 new nodes. Therefore, when $n=rf+r$ or $n=r(f+1)+1$, it is unnecessary to re-encode blocks with addition of one new node. The new node can synchronizes a partition of blocks from others to improve the availability of blocks without re-encoding. Hence, historical blocks are re-encoded for addition of a node when the number of old nodes is $n=r(f+1)+i$ $(i\ge 2)$. In summary, in terms of overall trends, the storage capability of BFT-Store is improved with more nodes.

In a permissioned setup, nodes typically belong in groups to organizations (e.g., companies trading with each other), where within an organization, there is trust between the nodes. Thus, fully replicating the state across organizations and simply partitioning the state within each organization would be a viable alternative. Then the storage consumption is $O(m)$, where m is the number of different organizations. In such setup, we can still apply our solution to this setting to further reduce the storage space. Basically, we can partition block data over all organizations via EC while the chunks of each organization can be distributed over nodes within the organization to reduce the storage overhead for each node.

7 Experimental Evaluation

We implement BFT-Store in Tendermint platform, which is an open source permissioned blockchain system adopted PBFT protocol for consensus and LevelDB[4] for data persistence. In BFT-Store, RS coding is implemented over a Galois Field with order 256, SHA256 is used to hash data, BN256 elliptic curve [30] is employed for digital signature and aggregate signature scheme is adopted to reduce the message size. At the network layer, each node maintains a TCP connection with its peers, so that all nodes can communicate with each other via P2P protocol. In order to conduct a

4. https://github.com/google/leveldb

TABLE 2
The Parameters in Experiments

Parameter	Value	Parameter	Value
# of machines	8	# of nodes n	$2 \sim 40$
faulty factor r	$3 \sim 5$	# of replicas c	$1 \sim 3$
block size	1 MB	space per node	200 MB
cache ratio	$0.1 \sim 1$	request distribution	UD&ZD

comprehensive evaluation, we also simulated nodes with Byzantine faults for test, which may keep silent with sending no message or send forged messages.

All experiments are conducted upon a cluster of 5 machines, where each node is equipped with 16 CPU cores with 2.10 GHz, 96 GB RAM and 3 TB disk space. The network bandwidth is 1 Gbps. To test the scalability, we run multiple (up to 8) Tendermint instances (nodes) on each machine to simulate a middle-size network with up to 40 nodes. Note that this testing way is also adopted in lots of latest works [12], [24]. We use built-in key-value store application of Tendermint for experiments. The results reported later focus on three key metrics: storage consumption, reading performance and re-encoding time. Table 2 lists the major parameters in our experiments.

7.1 Storage Consumption

Our first task is to evaluate the storage consumption of BFT-store. For the illustration purpose, we employ full-replication manner, the most-widely used strategy in blockchain systems and f-replication where each block is replicated over more than f nodes as described in Section 1, for comparison. For each block, the f-replication manner promises at least one correct node preserves it since there are at most f faulty nodes in system. First, we study the case where the block size is fixed to 1 MB, while the number of nodes varies from 2 to 40, as shown in Fig. 5. Fig. 5a reports the average storage consumption per block under different settings against the number of nodes. For the full-replication manner, the storage consumption per block increases linearly as more nodes attend, because each node stores a copy of entire blockchain. The storage occupation of f-replication is one third of full-replication, because each block just reserved by $f+1$ nodes. In comparison, in our manner, the storage consumption per block increases very slightly as more nodes join. For example, when $r=4$ and $c=3$, the storage consumption per block is 5.45 MB, even there are 40 nodes. Recall that r represents fault factor, and

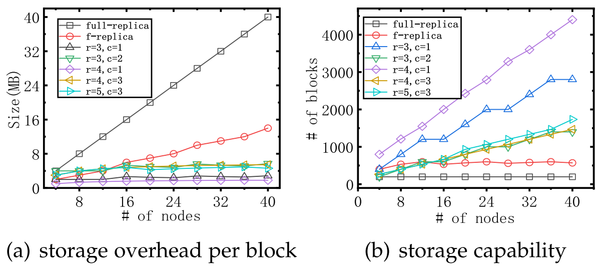


Fig. 5. Storage overhead with varying RS schema and number of nodes with fixed block size.

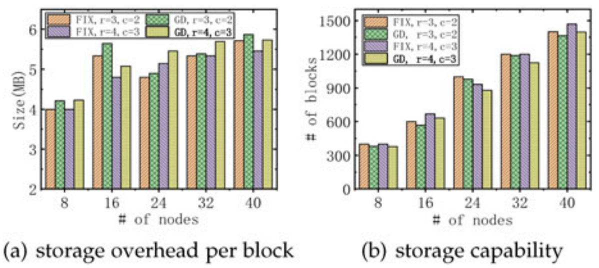


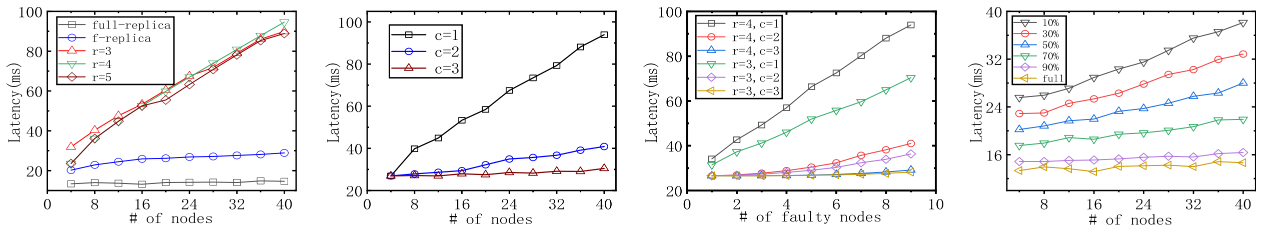
Fig. 6. Storage overhead with varying RS schema and number of nodes with variable block size.

c represents the number of replicas. Moreover, when fault factor r grows, the storage consumption per block will decrease due to the existence of more non-faulty nodes; when c rises, the storage consumption per block will increase because more replicas of each chunk are reserved. Fig. 5b shows the overall storage capability of the blockchain system. Assuming each node only owns 200 MB storage space, the overall storage capability is merely 200 blocks (1 MB per block) for the full-replication manner, no matter how many nodes are involved, and the storage capability of f-replication is triple of full-replication. On the contrary, the overall storage capability increases substantially when the network is enlarged. For example, when $r=4, c=1$, more than 4,400 blocks can be reserved in BFT-Store with 40 nodes.

We then evaluate the influence of variable block size to overall system performance, where the size of block follows a Gaussion distribution (GD) with mean 1 MB and standard deviation 0.1 MB, as shown in Fig. 6. Fig. 6a reports the average storage consumption per block with fixed or variable block size against the number of nodes. In comparison, with variable block size, there is a slight increase of storage consumption per block because the size of parity equals to that of the largest block in a coding round. For example, when $r=3, c=2$ and $n=40$, the average storage consumption per block is 5.71 MB and 5.86 MB with fixed and variable block size respectively. Fig. 6b shows the overall storage capability of BFT-Store under different settings. In blockchain system, the block size is closed to a fix value which is configured as maximal block size in advance. With variable block size, the storage capability increases slightly in comparison with fixed block size, which has an insignificant influence on BFT-Store. For example, when $r=4, c=3$, the storage capability is only dropped by 78 blocks with variable block size.

7.2 Read Performance

Our second task is to evaluate the efficiency of BFT-Store, i.e., the performance of reading blocks on two metrics: latency and throughput. To read a block, a client sends its request to a node randomly, and then it transmits back the block to client. To confirm the benefit of cache, requested block follows two distributions, Uniform and Zipfian distribution, where all blocks have the same probability of being accessed with Uniform distribution (UD) and some blocks are accessed more frequently under Zipfian distribution (ZD). We have verified that there is no significant benefit of cache under Uniformed distribution, and we just



(a) $c = 1$, UD (b) $r = 4$, UD (c) $r = 5$, UD (d) $r = 4$, $c = 2$, ZD

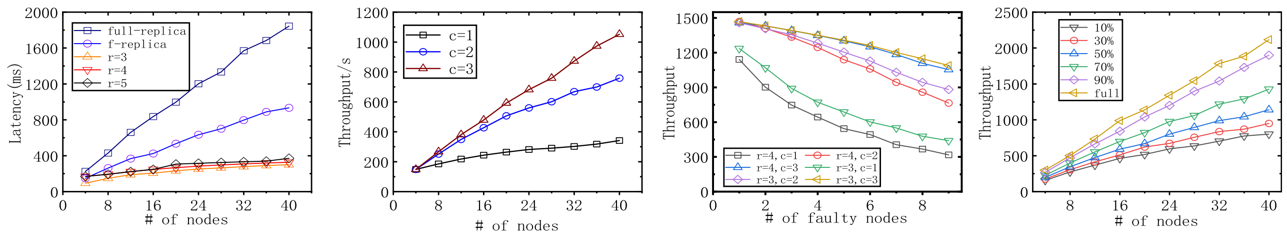
Fig. 7. Latency against number of nodes.

discuss the influence of cache for performance under Zipfian distribution.

Latency. First, we study the latency of BFT-Store against the number of nodes from 2 to 40 as shown in Fig. 7. The Uniform distribution is applied for requested blocks in Fig. 7a, 7b, and 7c while the Zipfian distribution is adopted in Fig. 7d. Fig. 7a reports the latency for block reading as r varies from 3 to 5 when $c = 1$. Note that the number of faulty nodes is set to $\lfloor\frac{n-1}{r}\rfloor$, and the faulty node always keeps silent or sends forged data. For full-replication manner, the latency keeps about 13.45 ms, no matter how many nodes are involved, due to no data transmission among nodes. The latency for f-replication manner is about double of full-replication, since the delegated node needs to pull the block from more than f nodes who store it. In comparison, the latency increases significantly in our manner as more nodes join BFT-Store because the amount of data transmitted for decoding increases. When $n = 40$ and $r = 4$, the latency for block access even reaches 94 ms. Fig. 7b reports the latency of BFT-Store when $r = 4$ while c rises from 1 to 3. As expected, with more replicas for each chunk, the latency of block access decreases because the probability $P_r(B)$ that block B is reachable grows. In Fig. 7b, when the number of nodes exceeds 10, there is an significant improvement of latency while c transfers from 1 to 2, because the network transmissions of chunks for decoding become the decisive factor. Fig. 7c shows the influence of malicious nodes to the latency under various cases, when $n = 40$. Since the maximal number of faulty number for $r = 3$ and $r = 4$ are $\lfloor\frac{40-1}{3}\rfloor = 13$ and $\lfloor\frac{40-1}{4}\rfloor = 9$, we just test up to 9 malicious nodes in system. As the number of faulty nodes grows, the latency increases significantly when $c = 1$ due to the rise of probability of decoding. Instead, the impact of latency is slight when $c = 3$ because more replicas drop the probability of decoding.

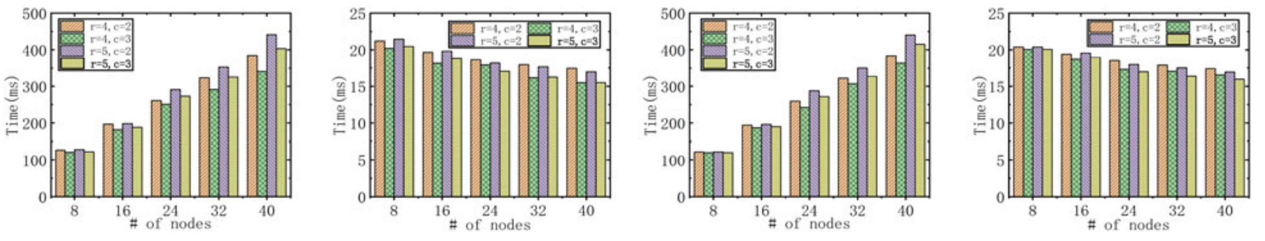
We adjust the parameters of Zipfian distribution so that the rate of cached blocks by each node out of all varies from 10 to 90 percent, while the case with full-replication manner is seen as the rate is 100 percent, and Fig. 7d reports the latency for block access with cache under different rate. When the rate rises, the latency decreases substantially since more requested block can be fetched from local LevelDB or memory. The overhead for loading block from LevelDB is much smaller than that for pulling chunks from remote nodes. When the rate is greater than 40 percent, the latency increases more slightly as the number of nodes grows, while it is very sensitive to the number of nodes if the rate is less than 40 percent.

Throughput. Second, we study the throughput of overall BFT-Store for block access against the number of nodes as shown in Fig. 8. Note that only the throughput of correct nodes is included and the results of faulty nodes are excluded. Fig. 8a reports the throughput of BFT-Store as r rises from 3 to 5 when $c = 1$. The throughput increases as the number of nodes grows under all five settings. Specifically, the throughput increases linearly with growth of nodes under full-replication and f-replication, while the improvement of throughput is not significant in our manner. For example, when $n = 40$, the throughput of full-replication and our manner are 1800+ and 400- respectively. With more nodes, the throughput of each node declines since the overhead for decoding rises. Thus the overall throughput increases very slightly even more nodes participates in system. Fig. 8b reports the throughput of BFT-Store when $r = 4$ as c varies from 1 to 3. As expected, the throughput of BFT-Store with more replicas is improved quickly, because the probability of decoding becomes smaller. When $c = 3$ and $n = 40$, the throughput of BFT-Store exceeds 1,000. Fig. 8c shows that the throughput of BFT-Store decreases as the number of faulty nodes grows when $n = 40$. Similarly, with more replicas for each chunk, the degradation of throughput is more slight. Particularly, the degradation is about 400 and 800 under $c = 3$ and $c = 1$ when the number of faulty node(s) grows from 1 to 9. Fig. 8d reports



(a) $c = 1$, UD (b) $r = 4$, UD (c) $r = 5$, UD (d) $r = 4$, $c = 2$, ZD

Fig. 8. Throughput against number of nodes.



(a) addition time per round (b) addition time per block (c) removal time per block (d) removal time per block

Fig. 9. Re-encoding time against number of nodes.

the throughput with cache under Zipfian distribution. The throughput for full-replication manner achieves 2,100, no matter how many nodes are involved because all blocks are stored locally. When the rate is 90 percent, the throughput of BFT-Store is about 1,900 with 40 nodes, which is very close to full-replication manner.

7.3 Re-Encoding

Last, we evaluate the re-encoding time of BFT-Store against the number of nodes as shown in Fig. 9. During re-encoding, the leader has to send out original blocks to all nodes for every coding round. Hence, its upload bandwidth may become the bottleneck. To relieve the stress of upload bandwidth of leader, we adopt the gossip network to implement the broadcasting in BFT-Store, where each node relays the received re-encode message to others randomly. By this method, the progress can be finished with time complexity $O(\log n)$. Fig. 9a reports the average re-encoding time per coding round under different cases when a new nodes adds. As expected, the time overhead per coding round increases as the number of nodes grows, since the number of re-encoded blocks per coding round increases. Observe that, when a chunk with more replicas, the time overhead decreases slightly, because the leader can collect $n-2f$ different chunks from fewer nodes earlier. For example, when $r=4$ and $4=20$, the re-encoding time per coding round is 384 ms under case $c=2$ while it declines to 341 ms under case $c=3$. Fig. 9b shows the average time overhead per block for re-encoding, and there is no significant rise of re-encoding time per block as the number of nodes grows. The time overhead for re-encoding increases with a larger fault factor r since leader has to recover original blocks for a coding round based on $(r-2)f+i$ $(i \geq 3)$ different blocks. Figs. 9c and 9d report the re-encoding time per round and block respectively when a existing nodes is removed from BFT-Store. The results is similar to addition of nodes, while the time consumption per round becomes a bit smaller since fewer nodes exists in BFT-Store.

8 Conclusion and Future Work

Full-replication manner without any scalability leads to the waste of huge storage, which is a serious problem in blockchain systems. The combination of PBFT with erasure coding provides a chance to partition blocks over all node in a hostile environment. Our contributions include: (i) exploit erasure coding for blockchain to reduce storage complexity per block to $O(1)$; (ii) propose a hybrid erasure coding and replicating to improve reading performance over encoded partition; (iii) design a re-encoding approach to deal with addition or removal of nodes. However, the RS coding employed in this study is not an efficient erasure coding. In the future, we will design repair pipelining mechanism in Byzantine environment to reduce repair time for decoding, which can improve the performance of BFT-store further.

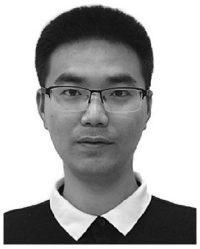
Acknowledgments

This work was supported in part by National Science Foundation of China (U1811264, U1911203, 61972152, and 61532021), Guangxi Key Laboratory of Trusted Software (kx202005), and ECNU Academic Innovation Promotion Program for Excellent Doctoral Students (YBNLTS2019-021).

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2019.
[2] M. Castro *et al.*, "Practical Byzantine fault tolerance," in *Proc. 3rd Symp. Operating Syst. Des. Implementation*, 1999, vol. 99, pp. 173–186.
[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.
[4] C. Burchert *et al.*, "Scalable funding of bitcoin micropayment channel networks," *Roy. Soc. Open Sci.*, vol. 5, no. 8, 2018, Art. no. 180089.
[5] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contract platform," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp., {NDSS}*, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-2_Al-Bassam_paper.pdf
[6] H. Dang *et al.*, "Towards scaling blockchain systems via sharding," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 123–140.
[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Princ.*, 2003, pp. 29–43.
[8] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proc. Int. Workshop Peer-to-Peer Syst.*, 2002, pp. 328–337.
[9] H. Chen *et al.*, "Efficient and available in-memory KV-store with hybrid erasure coding and replication," *ACM Trans. Storage*, vol. 13, no. 3, 2017, Art. no. 25.
[10] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, pp. 300–304, 1960.
[11] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," 2016.
[12] Y. Gilad *et al.*, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proc. 26th Symp. Operating Syst. Princ.*, 2017, pp. 51–68.
[13] Z. Yanchao, Z. Zhao, J. Cheqing, Z. Aoying, and Y. Ying, "SEBDB: Semantics empowered blockchain database," in *Proc. IEEE 35th Int. Conf. Data Eng.*, 2019, pp. 1820–1831.
[14] S. Wang *et al.*, "Forkbase: An efficient storage engine for blockchain and forkable applications," *Proc. VLDB Endowment*, vol. 11, no. 10, pp. 1137–1150, 2018.
[15] T. McConaghy *et al.*, "BigchainDB: A scalable blockchain database," *White Paper, BigChainDB*, 2016.
[16] T. T. A. Dinh *et al.*, "BLOCKBENCH: A framework for analyzing private blockchains," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1085–1100.
[17] C. Dwork *et al.*, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[18] Hyperledger, 2019. [Online]. Available: https://www.hyperledger.org
[19] N. Budhiraja *et al.*, "The primary-backup approach," *Distrib. Syst.*, vol. 2, pp. 199–216, 1993.
[20] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.
[21] C. Huang *et al.*, "Erasure coding in windows azure storage," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 15–26.
[22] K. Rashmi *et al.*, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster," in *Proc. 5th USENIX Workshop Hot Topics Storage File Syst.*, 2013, Art. no. 8.
[23] M. Silberstein *et al.*, "Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage," in *Proc. Int. Conf. Syst. Storage*, 2014, pp. 1–7.
[24] L. Luu *et al.*, "A secure sharding protocol for open blockchains," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 17–30.
[25] D. Schwartz *et al.*, "The ripple protocol consensus algorithm," *Ripple Labs Inc White Paper*, vol. 5, 2014.
[26] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, 1980.
[27] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *White Paper*, 2014.
[28] D. Boneh *et al.*, "A survey of two signature aggregation techniques," *RSA Cryptobytes*, vol. 6, no. 2, pp. 1–10, 2003.
[29] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong, "Performance analysis of private blockchain platforms in varying workloads," in *Proc. 26th Int. Conf. Comput. Commun. Netw.*, 2017, pp. 1–6.
[30] M. Naehrig, R. Niederhagen, and P. Schwabe, "New software speed records for cryptographic pairings," in *Proc. Int. Conf. Cryptology Inf. Secur. Latin Ame.*, 2010, pp. 109–123.



Xiaodong Qi received the BE degree in software engineering from Nanjing Normal University, Nanjing, China, in 2016. Currently, he is working toward the PhD degree supervised by Prof. Cheqing Jin. His research interests are in performance and scalability of blockchain systems, including BFT consensus protocols and blockchain sharding.

Zhao Zhang received the bachelor's degree in computer science from Northwest Normal University, China, in 2000, and the master's and PhD degrees from East China Normal University, China, in 2003 and 2012, respectively. She is an associate professor with East China Normal University (ECNU), China. Her research interests include distributed databases, blockchain, and location based service. Her works appeared in several major international conferences and journal on data management, including VLDB, ICDE, DASFAA, etc.

Cheqing Jin received the bachelor's and master's degrees from Zhejiang University, China, and the PhD degree in computer science from Fudan University, China, in 1999, 2002, and 2005, respectively. He is a professor with East China Normal University, China. Before joining East China Normal University, China on October 2008, he worked as an assistant professor with the East China University of Science and Technology, China. He is the winner of the Fok Ying Tung Education Foundation Fourteenth Young Teacher Award. He is a member of Database Technology Committee of China Computer Federation, and serves as a young associate editor of the *Frontiers of Computer Science*, an SCI journal. He has co-authored more than 80 papers, some of which received excellent paper awards, such as the Best Paper Award of the Chinese Journal of Computers, Best Paper Award of pervasive computing and embedding from the Shanghai Computer Society. His research interests include streaming data management, location-based services, uncertain data management, and sharing database management systems.

Aoying Zhou received the bachelor's and master's degrees in computer science from Sichuan University, China, and the PhD degree from Fudan University, China, in 1988, 1985, and 1993, respectively. He is the vice president of East China Normal University (ECNU), China, dean of School of Data Science and Engineering (DaSE), professor. He is the winner of the National Science Fund for Distinguished Young Scholars supported by the National Natural Science Foundation of China (NSFC) and the professorship appointment under the Changjiang Scholars Program of Ministry of Education (MoE). He is a CCF (China Computer Federation) fellow, the vice director of the Database Technology Committee of CCF, and associate editor-in-chief of the *Chinese Journal of Computer*. He served general chair of the ER2004, vice PC chair of ICDE2009 and ICDE2012, and PC co-chair of VLDB2014. His research interests include Web data management, data management for data-intensive computing, in-memory cluster computing and distributed transaction processing, and benchmarking for big data and performance.