MassBFT: Fast and Scalable Geo-Distributed Byzantine Fault-Tolerant Consensus

Zeshun Peng, Yanfeng Zhang, Tinghao Feng, Weixing Zhou, Xiaohua Li, Ge Yu
Northeastern University, China
{pengzeshun, fength, zhouwx}@stumail.neu.edu.cn, {zhangyf, lixiaohua, yuge}@mail.neu.edu.cn

***Abstract*—Geo-distributed consensus protocols provide high availability and resilience for distributed database services. These protocols group nodes by their data centers to leverage the network topology that spans across multiple data centers, thereby reducing costly cross-datacenter communication. However, they still face performance and scalability challenges due to inefficient log replication mechanisms. 1) These protocols rely on the leader node in each group to perform cross-datacenter log replication, creating a single-node performance bottleneck. 2) Byzantine receivers can behave arbitrarily, forcing the group leader to send multiple log copies during replication to prevent loss, thus causing redundant transmissions. 3) Since all groups must execute these logs in the same order, synchronizations across groups are necessary to maintain consistency when multiple groups are proposing concurrently, which also slow down log replication.**

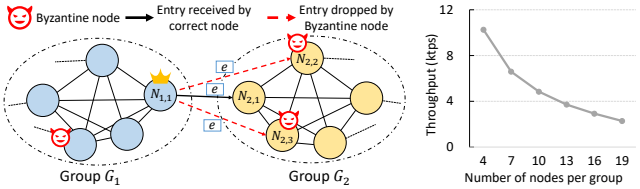
This paper presents `MassBFT`, a Byzantine fault-tolerant geo-consensus protocol that achieves high performance and scalability. We design an encoded bijective log replication to eliminate the leader bottleneck and reduce the cross-datacenter network consumption. We also propose asynchronous log ordering to eliminate synchronization across groups. Experimental results show that `MassBFT` is scalable, fault-tolerant, and outperforms state-of-the-art protocols with 5.49-29.96 times higher throughput under YCSB, SmallBank, and TPC-C workloads.

***Index Terms*—Replication, Fault Tolerance, Consensus**

I. Introduction

Byzantine fault-tolerant (BFT) consensus protocols are designed to provide availability and fault tolerance for distributed database services, including edge computing [1] [2], cross-border cooperation [3] [4], distributed robust graph storage [5], and authenticated query execution [6]. These services typically span multiple regions and data centers, handle large amounts of requests, and serve a widely distributed user base. For example, in edge computing [7] [8], such as healthcare [9], smart cities [10], and industry automation [11], edge servers are distributed across multiple geographic data centers and require mutual trust to maintain consistent database states. Geo-distributed consensus protocols have emerged as a scalable solution to meet the high demands of such applications.

Geo-consensus protocols, such as Steward [12], Blockplane [13], and GeoBFT [14], utilize high-speed local area networks (LAN) within data centers to reduce the costly wide-area network (WAN) communications between data centers. Instead of using all-to-all communication [15], these protocols adopt a hierarchical structure where each data center group elects a leader to handle log replication with other groups. Figure 1a shows an example of log replication, where group G_1 sends a log entry e to group G_2. This process involves three steps:



(a) (b)

Fig. 1. (a) An example of inter-group log replication. (b) Throughput under different group size settings.

1) **Batching**: The leader node $N_{1,1}$ of group G_1 creates an entry e containing a batch of transactions from local clients. 2) **Local Replication**: To ensure the integrity of entry e, $N_{1,1}$ broadcasts entry e within its group via LAN using local BFT consensus (e.g., PBFT [15] [16]). 3) **Global Replication**: As there are at most $\mathbf{f}$ indistinguishable Byzantine nodes in G_2 that may discard entry e, $N_{1,1}$ sends entry e to $\mathbf{f}+1$ nodes in G_2 to ensure that at least one *correct* node (i.e., $N_{2,1}$) receives it. Upon receipt, $N_{2,1}$ broadcasts e locally via LAN to ensure all correct nodes in G_2 receive it. The entries are then executed in a deterministic order across all nodes [17].

Geo-consensus protocols reduce WAN traffic at the expense of increasing LAN traffic. However, they still face significant scalability challenges due to limitations in leader-based log replication and inter-group synchronization, especially when cross-datacenter WAN transmissions become bottlenecks.

First, only the leader node broadcasts entries during global replication, creating a single-node bottleneck due to its limited upstream bandwidth. Although using multiple nodes in group G_1 to transmit an entry can alleviate this bottleneck, ensuring reliable transmission is still challenging. In Figure 1a, when using only G_1's leader node $N_{1,1}$ to send the entry, nodes in G_2 can easily identify any faulty behavior from $N_{1,1}$. However, when multiple sending nodes are used, pinpointing the cause of entry loss becomes challenging, as both faulty sending and receiving nodes can tamper with the entry (Section IV-A).

Second, in the broadcast example described above, massive redundant copies of entries are transmitted, resulting in high WAN traffic consumption. In Figure 1a, $N_{2,2}$ and $N_{2,3}$ in the receiver group G_2 are faulty. $N_{1,1}$ must send three copies of entry e to ensure that G_2 receives at least one correct copy. As the number of nodes in G_2 increases, the number of faulty nodes $\mathbf{f}$ that G_2 can tolerate also increases. Therefore, the number of copies $N_{1,1}$ sends increases linearly, causing the exhaustion of its upstream bandwidth. Figure 1b shows the scalability issue on GeoBFT [14], which employs the above replication strategy. We deploy 12 to 57 nodes across three

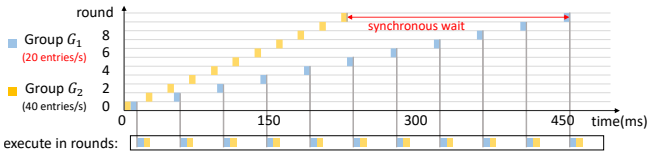


Fig. 2. G_2's performance is limited by G_1 due to the round-based ordering.

data centers in our nationwide cluster (detailed setup in Section VI), with each data center hosting 4 to 19 nodes. Each node is equipped with an independent WAN connection limited to 20 Mbps. The result shows a significant decrease in throughput as the number of nodes in each group increases.

Third, synchronizations are required when multiple group leaders propose their entries concurrently. Some protocols [18]–[20], such as Steward [12] and Blockplane [13], utilize Paxos [21] among all group leaders, allowing only one group leader to propose entries at the same time. Other protocols, such as GeoBFT and RCanopus [22], solve this single-leader bottleneck by allowing all group leaders to propose entries concurrently. To order the log entries proposed by different groups consistently, they adopted a predefined strategy that operates in rounds: in each round, every group must propose exactly one entry, and entries proposed in the same round are executed in order by comparing their group IDs.

This *synchronized* round-based ordering presumes the entry replication rates of all groups are fixed and identical. Unfortunately, even when the network between groups is stable, some groups may replicate entries slower than others due to limited WAN bandwidth or skewed workload distribution. As a result, entries proposed by the fast groups with a later execution order may arrive earlier. However, according to the predefined order, these entries can only be executed once all their predecessors have arrived and executed, limiting the throughput of the faster groups and increasing latency. In Figure 2, G_1 proposes 20 entries per second while G_2 proposes 40. In each round, entries from G_2 must wait for those from G_1 before they can be executed, thereby limiting G_2's performance.

To address these challenges, we propose `MassBFT`, a geo-consensus protocol that significantly enhances scalability.

We propose encoded bijective log replication for transmitting entries between groups, enabling all nodes to participate in the process, thus preventing group leaders from becoming single-node bottlenecks. In contrast to previous group sending approaches [14] [23] [24], where each node transmits complete entry copies (Section IV-A), we employ erasure coding [25] to reduce the number of entry copies transmitted between groups. Entries are divided into small *data chunks*, and additional *parity chunks* are encoded to ensure successful recovery in the event of data chunk loss due to node failures. By carefully designing a transfer plan, each node transmits only a few chunks instead of a complete entry copy while ensuring that the receiving group obtains enough chunks to reconstruct the entry, thereby reducing WAN traffic (Section IV-B).

We asynchronously determine the execution order of entries after they finish global replication, enabling entries proposed by faster groups to execute without delay. Each entry is assigned a vector timestamp (VTS) after completing global

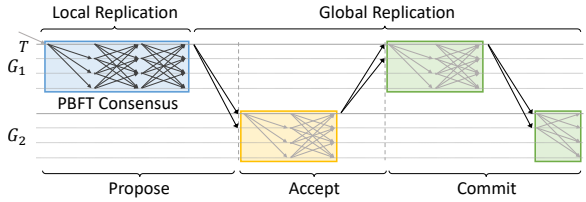


Fig. 3. Log replication of group G_1 in `Baseline`.

replication, and the *chronological order* of these VTSs dictates their execution order. Entries that are replicated earlier have smaller VTS and can be executed first, without waiting for other entries to finish replication. Replicating VTS is non-blocking and lightweight, adding negligible WAN traffic. Thus, even if a slower group has not yet received an entry, it can still replicate the entry's VTS on schedule (Section V-C). The VTSs are deterministically ordered to establish a consistent execution order among nodes (Section V-D).

In summary, the contributions of this paper include:

- We propose encoded bijective log replication to accelerate entry transmission. Group followers are used to send entries to avoid leader bottlenecks, while erasure coding encodes each entry into fault-tolerant chunks to reduce WAN traffic.
- We propose asynchronous log ordering to eliminate synchronization waiting issues before execution due to discrepancies in entry replication rates among groups.
- We present `MassBFT` based on the techniques above and conduct comprehensive evaluations. The results show that `MassBFT` is fast and scalable and outperforms all baselines.

II. Background on Geo-consensus

Geo-consensus protocols typically adopt a hierarchical architecture that groups nodes by data centers. They can vary in threat models (crash or Byzantine), structures (two or multi level), and protocols (PBFT [15] [16], Paxos [21], or Raft [30]) for local and global consensus. To clearly outline the workflow of geo-consensus protocols, we introduce `Baseline` as a generic model and review existing geo-consensus protocols.

A. Bottleneck Analysis in Log Replication

`Baseline` requires $\mathbf{n} \geq 3\mathbf{f} + 1$ nodes per group, with at most $\mathbf{f}$ out of $\mathbf{n}$ nodes that behave *faulty* (i.e., Byzantine). Since entire groups may crash due to WAN network partitions or data center power outages, `Baseline` also requires at least $\mathbf{n_g} \geq 2\mathbf{f_g} + 1$ groups to maintain liveness, allowing for up to $\mathbf{f_g}$ groups to crash simultaneously. For two groups, G_1 and G_2, Figure 3 shows the log replication of G_1 in `Baseline`. G_2's process is identical to G_1's and omitted. The detailed workflow of `Baseline` is described below.

Batching. As with any multi-master protocols [29] [22] [14], `Baseline` enables all groups to serve requests simultaneously. To minimize response time and WAN traffic consumption, clients send transaction requests to the nearest group leader based on network latency or physical distance [31] [12]. Once a certain number of requests are received or a timeout occurs, the group leader batches the transactions into a log entry (or simply, an entry). The entry also includes metadata such as *term* and *commitIndex* for global Raft consensus.

TABLE I
COMPARISON OF GEO-CONSENSUS PROTOCOLS WITH HIERARCHICAL ARCHITECTURE

Protocol	FT	Local consensus	Global consensus	Group failure	Log replication	Multi-master	Global ordering
D-Paxos [18]	CFT	Multi-Paxos	Paxos	$\mathbf{n_g} \geq 2\mathbf{f_g} + 1$	One-way (leader)	No	-
C-Raft [19]	CFT	Fast Raft	Fast Raft	$\mathbf{n_g} \geq 2\mathbf{f_g} + 1$	One-way (leader)	No	-
Steward [12]	BFT	PBFT	Paxos	$\mathbf{n_g} \geq 2\mathbf{f_g} + 1$	One-way (leader)	No	-
Blockplane [13]	BFT	PBFT	Paxos	$\mathbf{n_g} \geq 2\mathbf{f_g} + 1$	One-way (leader)	No	-
Ziziphus [20]	BFT	PBFT	Paxos	$\mathbf{n_g} \geq 2\mathbf{f_g} + 1$	One-way (leader)	No	-
ByzCoin [26]	BFT	-	PBFT	$\mathbf{n_g} \geq 3\mathbf{f_g} + 2$	Multi-level tree	No	-
Kauri [27]	BFT	-	HotStuff	$\mathbf{n_g} \geq 3\mathbf{f_g} + 1$	Multi-level tree	No	-
ByzCoinX [28]	BFT	-	PBFT	$\mathbf{n_g} \geq 3\mathbf{f_g} + 1$	Two-level tree	No	-
Canopus [29]	CFT	Raft	-	No	Leaf-only tree	Yes	Synchronous
RCanopus [22]	BFT	BFT+CFT	BFT	$\mathbf{n_g} \geq 3\mathbf{f_g} + 1$	Leaf-only tree	Yes	Synchronous
GeoBFT [14]	BFT	PBFT	-	No	One-way (leader)	Yes	Synchronous
`Baseline`	BFT	PBFT	Raft	$\mathbf{n_g} \geq 2\mathbf{f_g} + 1$	One-way (leader)	Yes	Synchronous
`MassBFT` (ours)	BFT	PBFT	Raft	$\mathbf{n_g} \geq 2\mathbf{f_g} + 1$	Encoded bijective	Yes	Asynchronous

Local Replication. PBFT is used for achieving local consensus within each group to tolerate Byzantine nodes. In the blue box of Figure 3, the group leader creates a certificate for the entry by running PBFT in three phases (pre-prepare, prepare, and commit) within its group. The certificate protects the entry from tampering by Byzantine nodes during the subsequent global replication. Once PBFT is complete, all correct nodes in the group receive the entry along with the certificate.

Global Replication. `Baseline` adopts Raft for global replication, with each group serving as a logical replica participating in Raft consensus. Since the Raft messages proposed by each group already undergo local consensus and are protected by PBFT certificates, Byzantine nodes cannot tamper with them. Therefore, the global Raft consensus aims to provide fault tolerance when an entire group crashes. As shown in Figure 3, the leader node of G_1 initiates Raft consensus (which proceeds through $propose$, $accept$, and $commit$ phases) by broadcasting the entry with the certificate to other groups G_2 through WAN. The leader of G_1 must send the entry to at least $\mathbf{f}+1$ nodes in G_2 to tolerate at most $\mathbf{f}$ Byzantine nodes in G_2 from dropping it. When the leader node of G_1 does not respond, G_2 can initiate a remote view change protocol [14] [20] to replace it. The correct nodes then forward the entry to all nodes within its group G_2 via LAN. When the leader of G_2 receives the entry, it creates an $accept$ message as a receipt. To prevent Byzantine decisions from the leader node, nodes in G_2 reach PBFT consensus on the $accept$ message before replying to G_1. As shown in the yellow box in Figure 3, PBFT consensus skips the prepare phase because nodes in G_2 do not need to agree on the consensus input (i.e., the entry), as it has already been certified by nodes in G_1 (details can be found in Ziziphus [20]). When the leader of G_1 receives $accept$ messages from $\mathbf{f_g}+1$ groups (i.e., G_2), it creates a $commit$ message to support these $accept$s and reaches PBFT consensus on $commit$ in G_1. After that, it broadcasts $commit$ to other groups and assumes that the entry is replicated. Nodes in G_2 commit the entry after receiving the $commit$ message.

Meanwhile, entries proposed by G_2 replicate to G_1 in a similar way, as all groups accept requests concurrently. To order the entries proposed by G_1 and G_2 consistently across all nodes (a consistent execution order is crucial for accurate results), `Baseline` adopts a round-based synchronous ordering that proceeds in rounds. Every group can propose exactly one entry per round. After receiving all entries in a round, a node orders these entries by the IDs of the groups that created them and executes these entries in this order.

B. Comparison of Geo-Consensus Protocols

Table I compares existing geo-consensus protocols with `Baseline` mentioned above and our protocol, `MassBFT`.

Consensus Architecture. The choice of the local consensus protocol highly depends on their failure models. D-Paxos [18], Canopus [29], and C-Raft [19] only tolerate crash failures because they use crash fault-tolerant (CFT) protocols (i.e., Multi-Paxos [32], Raft, and Fast Raft [19]) for local consensus. Canopus groups nodes within the same rack to optimize the performance of local consensus. C-Raft uses Fast Raft to reduce the number of network round-trips (RTTs) during consensus. In contrast, Steward, Blockplane [13], Ziziphus, and GeoBFT use PBFT for local consensus to tolerate Byzantine failures. RCanopus [22] is similar to that of Canopus. Nodes within the same rack are organized into CFT sub-groups. Each group, consisting of multiple geographically close sub-groups, uses BFT consensus to tolerate Byzantine sub-groups.

The goal of global consensus is to reduce network traffic between groups. There are three types of global consensus protocols: 1) Protocols like D-Paxos and Steward use CFT global consensus, which can tolerate up to $\lfloor(\mathbf{n_g}-1)/2\rfloor$ crashed groups $\mathbf{f_g}$ at the same time. 2) Protocols like ByzCoin [26] and RCanopus use BFT global consensus, which can cope with Byzantine groups. Moreover, ByzCoin, Kauri [27], and ByzCoinX [28] do not employ local consensus. The group leaders serve as intermediaries, disseminating the proposal from the PBFT leader to their group members and aggregating their responses back to the PBFT leader. 3) GeoBFT and Canopus replicate logs directly without using global consensus. After achieving local consensus, the entry is directly broadcast to other groups. This approach can significantly reduce latency but at the cost of sacrificing group fault tolerance (i.e., they cannot tolerate Byzantine or crash groups).

Log Replication. Most existing protocols [12]–[14] adopt a one-way replication strategy where only group leaders initiate entry transfer. Moreover, they must send entry copies to additional receivers to avoid entry loss during replication. For example, the group leaders in Steward send each entry to all nodes, while GeoBFT reduces WAN overhead at the cost of

additional intra-group communication, where the entry is only sent to $\mathbf{f}+1$ nodes within each group across all other groups. ByzCoin and Kauri employ multi-level trees to reduce WAN transmission overhead at the expense of latency. Additionally, Kauri employs HotStuff [33] to reduce message complexity and leverages pipelining techniques to enhance throughput. However, HotStuff may experience latency bottlenecks in geo-distributed scenarios. ByzCoinX employs a two-level tree to balance the latency and the inter-group network traffic. Canopus and RCanopus utilize a Leaf-only Tree overlay for entry replication, which supports pipelining but also increases latency due to multiple rounds of entry exchanges.

Ordering. Steward and Blockplane limit only one group to propose entries at a time. For example, in Steward, when multiple group leaders propose entries at the same time, only one (or none) can succeed because Paxos can only reach a consensus on a single entry at a time. Therefore, the execution order of entries is exactly the output of the global consensus. D-Paxos reduces contention in Paxos by using a round-robin strategy, where group leaders take turns acting as the global consensus leader to propose entries. However, D-Paxos still allows only one group leader to propose entries at a time.

In contrast, Canopus, RCanopus, and GeoBFT enable all groups to propose entries simultaneously. When multiple groups propose entries at the same time, nodes may receive these entries in any order. To establish a global consistent execution order, these protocols operate in rounds to order these entries. In each round, every group proposes exactly one entry. Execution at a node begins once all entries for the current round have been received. However, this method requires all groups to replicate entries at the same rate.

III. MassBFT Overview

A. System Model

`MassBFT` is deployed on geo-distributed data centers. Nodes within the same group are on the same data center and are connected through a fast data center network (LAN). Each node also has an exclusive public network (WAN) with limited bandwidth. We denote the i-th group as G_i, and the j-th node in the i-th group G_i as $N_{i,j}$.

Threat Model. `MassBFT` adopts the BFT model, where *faulty* nodes can behave arbitrarily. A strong adversary can control all faulty nodes but cannot break the cryptographic primitives. `MassBFT` adopts a public-key infrastructure (PKI), where each node has a public-private key pair for signing and verifying messages. We denote the number of faulty nodes in group G_i as $\mathbf{f_i}$, and the number of nodes $\mathbf{n_i}$ in group G_i is at least $3\mathbf{f_i}+1$. The entire group is crash-only, and `MassBFT` requires $\mathbf{n_g} \geq 2\mathbf{f_g}+1$ groups, with up to $\mathbf{f_g}$ groups crashed simultaneously.

Network Model. For *liveness*, `MassBFT` assumes the partial synchrony model [34]. The network is considered to be reliable most of the time but may also experience periods of asynchrony. During each unstable period, a global stabilization time (GST) exists, after which all transmissions between correct nodes arrive within a known bound Δ.

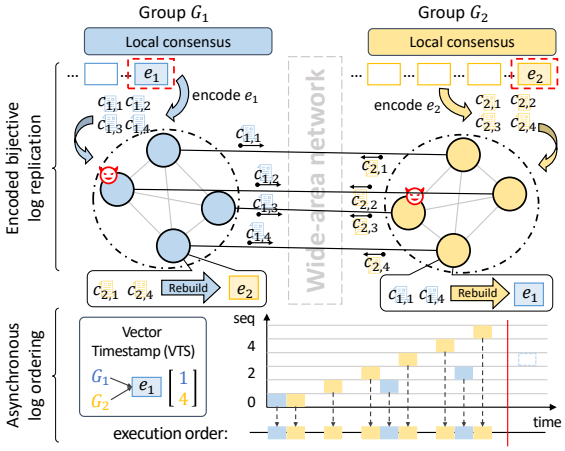


Fig. 4. Overview of `MassBFT`.

B. Protocol Overview

Figure 4 shows the workflow of `MassBFT`. The local replication is the same as that of `Baseline` (Section II-A). During global replication, entries are transferred via encoded bijective log replication, followed by asynchronous log ordering to establish a consistent execution order. The key features of `MassBFT` are described below.

Multi-master Replication with Hierarchical Architecture. In `MassBFT`, all groups concurrently serve requests and propose entries. First, entries containing clients' transactions undergo local PBFT consensus within the group to mask the behaviors of faulty nodes. Then, the entries are replicated among all groups using Raft to tolerate crashed groups. After Raft consensus, entries generated by the same group are already in order (proven in Section V-D). For entries generated by different groups, `MassBFT` adopts the asynchronous log ordering to ensure a consistent execution order across nodes, preventing slow groups from slowing down others.

Bijective Low-redundancy Log Replication. `MassBFT` utilizes all nodes to transmit entries and employs erasure coding to minimize redundancy. We now consider sending an entry between two groups, as broadcasting entries to all groups can be built from it. In Figure 4, after achieving local consensus, every correct node in group G_1 has a copy of entry e_1. Each node in G_1 then deterministically splits e_1 into data chunks, and encodes additional parity chunks to tolerate chunk loss caused by faulty nodes in both groups Each chunk is sent only once from one node in G_1 to another in G_2 based on a transfer plan to reduce redundancy. Finally, correct nodes in G_2 exchange their received chunks, ensuring each node has sufficient chunks to rebuild entry e_1. The replication process of e_2 is similar than that of e_1. Notably, only the *propose* phase, which handles entry replication, is enhanced with encoded bijective log replication, while the *accept* and *commit* phases remain the same as `Baseline`.

Asynchronous Ordering with Vector Timestamps. To tolerate slow groups, `MassBFT` orders entries by their logical vector timestamps (VTS), which are assigned after they finish global replication. To avoid centralized timestamp assignment,

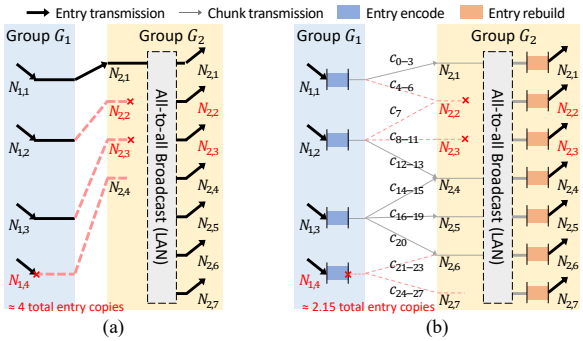


Fig. 5. An example of bijective log replication. (a) The general approach, where every node in G_1 sends a *complete* entry copy to G_2; (b) The erasure-coded approach, where each node in G_1 sends 7 chunks and each node in G_2 receives 4 chunks, with each chunk having a size of $1/13$ the entry size.

MassBFT uses logical rather than physical timestamps. The entry replication and order determination are asynchronous. Each entry is first assigned a VTS after completing replication. A consistent execution order is then asynchronously established by comparing these VTSs element-wise. If an entry (proposed by a fast group) finishes replication earlier than expected, it will have a smaller VTS and thus be ordered earlier.

IV. Encoded Bijective Log Replication

To highlight the efficiency of encoded bijective log replication, first, we introduce a general approach of bijective group sending [14] [23] [24] and discuss its limitations. Then, we propose the erasure-coded approach to reduce its complexity.

A. General Approach

The general approach is proposed by [23] [24] and adopted by GeoBFT in its remote view-change phase. Its core idea is to mask the failures caused by $\mathbf{f_1}$ faulty sending nodes in G_1 and $\mathbf{f_2}$ faulty receiving nodes in G_2. Specifically, each group selects $\mathbf{f_1}+\mathbf{f_2}+1$ nodes, and each node in G_1 sends the entry e to different nodes in G_2. Despite simultaneous failures of $\mathbf{f_1}$ nodes in G_1 and $\mathbf{f_2}$ nodes in G_2, there are still $\mathbf{f_2}+1$ nodes in G_1 sending e, which is sufficient to ensure that at least one correct node in G_2 receives e. Then, the correct node broadcasts e locally so that all nodes in G_2 receive e.

Figure 5a shows an example where G_1 and G_2 have 4 and 7 nodes, respectively. Since entry e is protected by the certificate generated during local PBFT consensus, faulty nodes can only drop entry e but cannot tamper with it (discussed in Section II-A). Each of the four nodes in G_1 sends a copy of entry e to a distinct node in G_2. Because node $N_{1,4}$ in G_1 and nodes $\{N_{2,2}, N_{2,3}\}$ in G_2 are faulty, only $N_{2,1}$ receives e from $N_{1,1}$. Therefore, other nodes in G_2 can still receive e from $N_{2,1}$.

Discussion. The bijective sending approach effectively reduces the leader bandwidth but is applicable only when the group sizes are *nearly the same* (i.e., the sum of f_1+f_2+1 is not less than the size of either G_1 or G_2). In cases where there are significant differences in the number of nodes across groups, a lower bound on communication costs (which is greater than f_1+f_2+1) and an optimal sending plan can be determined using the partitioned bijective sending [23] [24]. However, all these sending approaches require a node to send or receive complete entry copies. The overhead can be further reduced by employing erasure codes.

B. Erasure-Coded Approach

Erasure coding [35] is an error correction technique that improves the reliability and availability of message transmission [36] [37]. A widely used erasure code is the Reed-Solomon (RS) code [25], which encodes messages into a total of n_{total} chunks, consisting of n_{data} data chunks and n_{parity} parity chunks where $n_{\text{parity}} = n_{\text{total}} - n_{\text{data}}$. The parity chunks are encoded as redundancy for the data chunks. In other words, any n_{data} out of n_{total} chunks can be used to rebuild the original message. Notably, the message can only be rebuilt if all input chunks are correct. Attempting to rebuild with corrupted or out-of-order chunks will result in an erroneous message.

Our approach encodes entries into smaller chunks and distributes them evenly among all sender group nodes for concurrent transfer. Nodes in the receiver group receive these disjointed chunks and exchange them through LAN to rebuild the original entry. As a result, the total WAN traffic is reduced to the size of the total chunks and is evenly distributed among all nodes. To maximize the use of network resources, one option is to allow faster nodes to send more chunks. However, since all nodes have an equal probability of being faulty, this approach would require encoding additional parity chunks to prevent the worst-case scenario when these nodes are faulty, thereby increasing overall WAN traffic.

In MassBFT, each node within a group sends the same number of chunks based on a predetermined transfer plan. As shown in Algorithm 1, the transfer plan is a list of tuples $\langle c, i, j\rangle$, denoting the transfer of chunk c from node with ID i in sender group G_1 to node with ID j in receiver group G_2. The total number of chunks n_{total} is computed by finding the least common multiple (LCM) of the number of nodes ($\mathbf{n_1}$ and $\mathbf{n_2}$) in G_1 and G_2. To evenly distribute the workload across nodes, each node in G_1 sends $nc_1 = n_{\text{total}}/\mathbf{n_1}$ chunks, while each node in G_2 receives $nc_2 = n_{\text{total}}/\mathbf{n_2}$ chunks (lines 1-3). This ensures that each chunk is sent and received only once.

The number of parity chunks n_{parity} must also be determined to ensure that entries can be successfully rebuilt in the event of data chunk loss. Similar to Section IV-A, two types of chunk loss occur during erasure-coded replication: **1)** faulty nodes in G_1 may send fake chunks to nodes in G_2, and **2)** faulty nodes in G_2 may broadcast fake chunks locally. In the worst case, the chunks sent by faulty nodes in G_1 and the chunks received by faulty nodes in G_2 are disjoint sets. Therefore, the maximum number of lost chunks n_{parity} is determined by the number of faulty nodes ($\mathbf{f_1}$ and $\mathbf{f_2}$) in G_1 and G_2, as well as the number of chunks each faulty node sent or received (nc_1 and nc_2) (lines 4-5), while the remaining chunks are data chunks n_{data}.

The chunks are assigned to nodes in ascending order of their IDs. For a node with ID i in the sender group G_1, it first finds the IDs of the chunks to be sent (lines 7-8) and determines their corresponding receiver IDs (lines 9-10). Similarly, if the node acts as a receiver, it finds the chunk IDs to be received and calculates the corresponding sender IDs (lines 11-14).

Algorithm 1: Transfer Plan Generation
(for each sender-receiver group pair)

Input: node with ID i (in the sender group or receiver group), sender group G_1 of size $\mathbf{n_1}$, and receiver group G_2 of size $\mathbf{n_2}$.
Output: the number of total chunks n_{total} and data chunks n_{data}. The transfer plan $plan$ with a list of tuples $\langle$chunk ID, sender node ID, receiver node ID$\rangle$.

$n_{\text{total}} \leftarrow \texttt{LCM}(\mathbf{n_1}, \mathbf{n_2})$; // calculate the least common multiple
$nc_1 \leftarrow n_{\text{total}}/\mathbf{n_1}$; // number of chunks a node sends in G_1
$nc_2 \leftarrow n_{\text{total}}/\mathbf{n_2}$; // number of chunks a node receives in G_2
$\mathbf{f_1} \leftarrow \lfloor(\mathbf{n_1}-1)/3\rfloor$; $\mathbf{f_2} \leftarrow \lfloor(\mathbf{n_2}-1)/3\rfloor$; // faulty nodes
$n_{\text{parity}} \leftarrow nc_1 \times \mathbf{f_1} + nc_2 \times \mathbf{f_2}$; // number of chunks lost
$n_{\text{data}} \leftarrow n_{\text{total}} - n_{\text{parity}}$; // number of chunks delivered
if *node with ID i is in the sender group G_1* **then**
  for *chunk with ID c* **from** $nc_1 \times i$ **to** $nc_1 \times (i+1) - 1$ **do**
    $j \leftarrow \lfloor c/nc_2 \rfloor$; // c, i, j start from 0
    Add $\langle c, i, j\rangle$ to $plan$; // send chunk c to node j in G_2
else // node with ID i in the receiver group G_2
  for *chunk with ID c* **from** $nc_2 \times i$ **to** $nc_2 \times (i+1) - 1$ **do**
    $j \leftarrow \lfloor c/nc_1 \rfloor$;
    Add $\langle c, j, i\rangle$ to $plan$; // receive c from node j in G_1
return n_{total}, n_{data}, and $plan$;

Case Study. Figure 5b shows an example where a 4-node group G_1 sends an entry to a 7-node group G_2. The total number of chunks $n_{\text{total}} = 28$ is determined by the LCM of 4 and 7. Each G_1 node sends 7 chunks, while each G_2 node receives 4 chunks. Considering 1 faulty node in G_1 and 2 faulty nodes in G_2, the maximum potential chunk loss is $1 \times 7 + 2 \times 4 = 15$, matching the number of parity chunks n_{parity}. Based on the transfer plan, correct G_2 nodes receive a total of 13 correct chunks (indicated by gray lines) and broadcast these chunks via LAN. Finally, each G_2 node can obtain at least 13 chunks, which is sufficient to rebuild the original entry. In the example above, the encoded bijective approach sends a total of $n_{\text{total}}/n_{\text{data}} \approx 2.15$ entry copies, which is significantly lower than the bijective-only approach (Figure 5a), which is 4.

C. Optimistic Entry Rebuild

During chunk transmission, faulty senders may send fake chunks to receivers, while faulty receivers may also broadcast fake chunks within their groups. The fake chunks are indistinguishable from the correct ones, as they do not undergo local consensus and produce a certificate. Since erasure coding requires all input chunks be correct and properly ordered, receivers might need to rebuild the entry multiple times. This repeated process can significantly affect the liveness property.

We adopt an optimistic approach to validate the received chunks, inspired by previous studies [38]–[40]. This approach leverages Merkle trees and Merkle proofs [41] [42] to efficiently verify whether the received chunks are encoded from the same entry. A Merkle tree is an authenticated data structure used to verify a large data set (i.e., a set of chunks). Each leaf node of the Merkle tree contains the hash of an individual data block (i.e., a chunk), and each internal node contains the hash of its two child nodes. These data blocks can be represented by a single Merkle root, which is the root hash

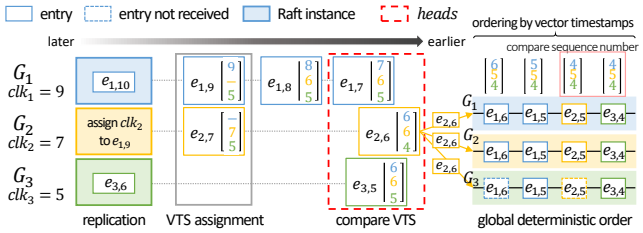


Fig. 6. The workflow of asynchronous log ordering.

obtained by recursively hashing the child nodes. A Merkle proof can efficiently prove whether a data block is included in the data set. The proof consists of node hashes on the path from the leaf node to the Merkle root.

After encoding an entry into chunks, each sender node constructs a Merkle tree from these chunks and sends the entry's PBFT certificate, along with the chunks and their Merkle proofs, to the corresponding receivers. The receivers group these chunks by their Merkle roots into buckets for classification. Tempered chunks must not be grouped into the same bucket as the correct ones. This is because chunks sharing the same Merkle root are encoded from the same entry. When a chunk is tempered, its Merkle root must change.

When the number of chunks in a bucket reaches the threshold (i.e., n_{data}), the receiver tries to rebuild the entry and validates it with its certificate. If the validation fails, indicating that all chunks in the bucket are fake (as they share the same Merkle root), the receiver logs the IDs of these chunks (inferred from their Merkle proofs). It no longer accepts any further chunks with these IDs to prevent denial-of-service (DoS) attacks. If the entry passes validation, the receiver accepts it and proceeds with the Raft consensus process.

The *liveness* and *safety* properties are discussed as follows:

Liveness. The encoded bijective log replication ensures the liveness of inter-group replication [28] [6]. When the network is temporarily unstable and entries are delayed or lost, if the upper protocol provides liveness (e.g., using Raft as global consensus), no additional measures are required. Otherwise (e.g., in GeoBFT), nodes can apply a reliable transmission protocol like TCP to ensure eventual delivery.

Safety. Although faulty nodes may tamper with some chunks, as discussed in Section IV-C, the rebuilt erroneous entries can be identified by verifying their PBFT certificates.

V. Asynchronous Log Ordering

In `MassBFT`, the number of nodes in each group may differ due to nodes joining or leaving [2]. Since the entry sending rate of a group is related to its size (verified in Section VI-D) and workload, different groups may propose entries at varying and changing rates. The round-based ordering failed to obtain a flexible execution order, thereby limiting throughput to the pace of the slowest group. Therefore, it is necessary to establish a new total order to execute entries more efficiently.

A. Protocol Design

Similar to other hierarchical protocols [12] [13], each group in `MassBFT` participates in the global Raft consensus as a logical replica. As `MassBFT` adopts a multi-master architecture,

there are a total of n_g global Raft instances running in parallel. A group G_i serves as the leader of the i-th Raft instance (denoted as G_i's Raft instance) and participates in all other Raft instances as followers. We denote the entry generated by group G_i with local sequence number m as $e_{i,m}$.

Entry Replication. An entry e first achieves PBFT consensus within the sending group G_i. Then, G_i sends e to all other groups using the encoded bijective replication. Concurrently, G_i initiates a global Raft consensus through its own Raft instance, where it acts as the leader. The message includes the entry digest and the corresponding PBFT certificate.

Vector Timestamp Assignment. As shown in Figure 6, each group G_i maintains an independent local logical clock, clk_i, to timestamp entries. Each entry is assigned a *vector timestamp* (VTS) during the VTS assignment. The global order of entries is determined by comparing their VTSs (Section V-D).

A VTS consists of a set of timestamps. When an entry e achieves global Raft consensus, each group G_i assigns the time from its local clock clk_i to the i-th element of e's VTS, denoted as $e.VTS[i]$. This timestamp is then replicated across G_i's Raft instance for consistency by piggybacking it onto the Raft consensus during entry replication. Additionally, to ensure liveness, a group increments its local clock once the entry it has proposed completes the global Raft consensus.

For example, Figure 7a shows that after entry $e_{1,10}$ achieves Raft consensus, the proposer group G_1 increases its local clock clk_1 from 9 to 10. After receiving entry $e_{1,10}$ from G_1's Raft instance, the three groups, G_1, G_2, and G_3 assign their local clock times, 10, 7, and 5 to $e_{1,10}$. The groups then exchange their timestamps through their respective Raft instances. Finally, every group possesses a copy of entry $e_{1,10}$ and its VTS $\langle 10, 7, 5\rangle$.

We overlap an entry's replication and VTS assignment to reduce latency (Section V-B). We also enhance this process to prevent faulty (or slow) groups and nodes from delaying or blocking the assignment of VTSs (Section V-C).

Ordering by Vector Timestamp. Each entry is assigned a complete VTS, allowing us to establish their global order by comparing their VTSs (Lemma V.4). However, since entries with VTSs arrive out of order in a streaming manner, additional measures are needed to predict the next entry to execute.

Entries proposed by the same group G_i (e.g., $e_{1,7}$, $e_{1,8}$, and $e_{1,9}$ by G_1 in Figure 6) are ordered by their local sequence numbers, since their VTSs increase monotonically (Lemma V.5). Therefore, among the unexecuted entries proposed by group G_i, entry $head_i$ with the smallest local sequence number (e.g., $e_{1,7}$) must precede than the others (e.g., $e_{1,8}$ and $e_{1,9}$). As a result, the next entry to execute can be determined by finding the most preceding entry in $heads = \langle head_1, ..., head_{\mathbf{n_g}}\rangle$ (e.g., $e_{1,7}$, $e_{2,6}$, and $e_{3,5}$). Therefore, a node can recursively select the entry with the smallest VTS in $heads$ for execution. By default, the VTSs of all entries in $heads$ must be assigned completely to determine the next entry to execute. To minimize the blocked waiting caused by assigning VTS, a node can infer

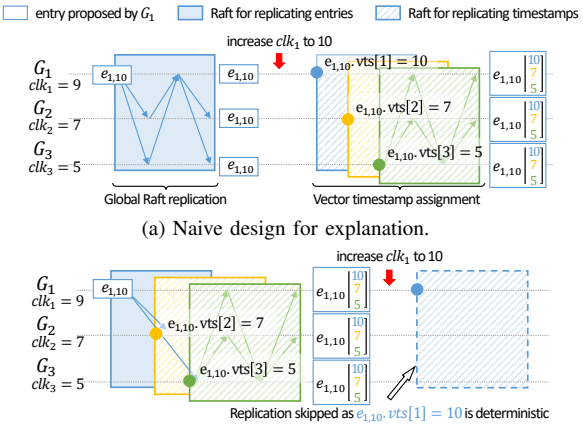


Fig. 7. An illustrative example of entry replication and vector timestamp assignment of entry $e_{1,10}$.

the possible VTSs of entries in $heads$ before they are proposed or before their VTS assignment is finished (Section V-D).

Moreover, concurrent replication may result in entries from different groups having identical VTSs ($e_{2,5}$ and $e_{3,4}$). A consistent total order is still achievable by comparing their local sequence numbers seq and group IDs gid (Section V-D).

B. Overlapped Vector Timestamp Assignment

As shown in Figure 7a, the VTS assignment discussed above requires two consecutive rounds of Raft consensus, one for replicating the entry and the other for replicating the timestamps assigned to it. The two phases perform serially, causing high consensus latency (about 3 RTTs).

As shown in Figure 7b, it is possible to reduce latency to 2 RTTs while maintaining correctness by overlapping these two phases: **1)** Group G_i skips assigning the time from its local clock clk_i to entries $e_{i,n}$ proposed by itself, because the assignment $e_{i,n}.vts[i] = n$ is deterministic. Once $e_{i,n}$ has achieved Raft consensus, G_i advances its local clock clk_i to n. **2)** Every other group G_j directly assigns the time from its local clock clk_j to $e_{i,n}$ on receiving $e_{i,n}$ via Raft's *propose* message from G_i, instead of waiting for $e_{i,n}$ to achieve Raft consensus. However, overlapping the replication phase and the VTS assignment phase of entry e potentially leads to an exception where some of e's VTS assignment succeeds but e's replication fails. In this scenario, the correct groups cannot receive e and therefore cannot execute entries after it, creating blocking issues. However, this exception cannot occur, as proven by the following lemma:

Lemma V.1. ***(atomicity)*** *If entry e completes replication, all of its timestamps $e.vts[j]$ will eventually be assigned and replicated. Otherwise, none of its timestamps $e.vts[j]$ will complete replication.*

Proof. When the timestamp $e_{i,n}.vts[j]$ from group G_j is received by group G_k in a *propose* message, G_k will respond with an *accept* message if it has received the entry $e_{i,n}$. Otherwise, G_k will not reply until receiving the actual entry $e_{i,n}$ from group G_i, or it can request the entry $e_{i,n}$ from G_j if group G_i crashes. Thus, if the timestamp $e_{i,n}.vts[j]$ completes

replication, at least $\mathbf{f_g} + 1$ groups must have received $e_{i,n}$ and replied *accept* to G_j, meaning $e_{i,n}$ is globally replicated. □

C. Handling Faulty Groups and Nodes

Slow Sender Groups. We define a slow sender group as a group that proposes and replicates messages at a slower rate than other groups. Although a slow group may take a longer time (tens of milliseconds, depending on the message size and its bandwidth) to send each message, when the network is reliable (due to partial synchrony network assumption), all messages sent by the group can achieve consensus in time.

Lemma V.2. ***(liveness)*** *After receiving entry $e_{i,m}$, group G_j can assign the current time from its clock clk_j to $e_{i,m}.vts[j]$ and finish replicating $e_{i,m}.vts[j]$ within a short, finite time t.*

Proof. When group G_j assigns the time from its clock clk_j to entry $e_{i,m}$, another entry $e_{j,n}$ proposed by G_j itself may still be replicating. Therefore, the timestamp $e_{i,m}.vts[j]$ must wait for the replication of $e_{j,n}$ to complete because they use the same consensus instance. However, the replication of $e_{i,m}.vts[j]$ cannot be blocked by $e_{j,n}$ for a long time, because entry $e_{j,n}$ proposed by G_j can reach consensus in a short and finite time t (as previously discussed). We also utilize the pipelining technique to accelerate this process. □

Slow Receiver Groups. A slow receiver group G_j receives entries $\langle e_{i,1}, ..., e_{i,n} \rangle$ slower than the proposing group G_i sends them. Since these entries must be received in order, G_j cannot receive and assign its local clock time to the latest entry $e_{i,n}$ in time. Recall that during Raft consensus, a follower group G_k replies with an $accept$ message to the leader group G_i when receiving entry $e_{i,n}$ from G_i in a $propose$ message. To notify the slow group G_j that $e_{i,n}$ has completed replication, G_k also broadcasts the $accept$ message directly to other groups (i.e., G_j) without using Raft consensus. The slow group G_j assigns its local clock time to $e_{i,n}.vts[j]$ on receiving $accept$ messages from $\mathbf{f_g}+1$ groups (via direct broadcast). This approach avoids slowing down entry ordering of other groups.

Crashed Groups. When a group G_i crashes, its Raft instance (i.e., the Raft instance led by G_i) will elect a new leader G_j. Since G_j has received the latest log of G_i's Raft instance, it can determine G_i's clock time $clk_i = n$ from the local sequence number of the latest entry $e_{i,n}$ proposed by G_i. G_j then represents G_i by assigning timestamps to entries with G_i's clock time (i.e., $e.vts[i] = n$) and replicates $e.vts[i]$ using G_i's Raft instance. Since G_i has crashed and cannot propose entries, clk_i will not increase. When G_i recovers later, G_j transfers the leadership of G_i's Raft instance back to G_i, and G_i can serve requests normally.

Byzantine Nodes. Faulty followers are masked via local PBFT consensus, and a faulty leader can be detected by verifying its proposal. For example, if the leader node proposes a stale timestamp, since its local clock must be monotonically non-decreasing, the proposal cannot be validated by local follower nodes and obtain a valid certificate signed by $2\mathbf{f} + 1$ nodes in its group (detailed in Section II-A).

The leader node of a group may exhibit Byzantine behavior by intentionally delaying the sending of Raft messages, while ensuring the delays are shorter than the Raft timeout threshold, reducing consensus performance. To prevent this behavior, we can allow additional $\mathbf{f}$ group followers to send the messages (i.e., message flooding [43] [44]), as the message has reached local consensus and each node in this group has a copy of it.

Theorem V.3. *After the replication of entry e is completed, a complete VTS (with each element been properly set) will be assigned to the entry e within a finite time t.*

Proof. Lemma V.1 ensures that all replicated entries will be assigned complete VTSs. Lemma V.2 and the failure-handling mechanisms guarantee that these VTSs will complete replication quickly, within a finite time t. □

D. Deterministic Ordering

Differing from vector clocks [45], which are used to determine causal orders, the comparison of VTSs is conducted *element-wise* and terminates once a discrepancy between elements is found. This method is similar to the lexicographical order used in dictionaries. For entries with identical VTSs, we further order them by their local sequence numbers seq and group IDs gid. For example, in Figure 6, entry $e_{2,6}$ with VTS $\langle 6, 6, 4 \rangle$ is ordered before $e_{3,5}$ with VTS $\langle 6, 6, 5 \rangle$, because $e_{2,6}.vts[1] = e_{3,5}.vts[1] = 6$ and $e_{2,6}.vts[2] = e_{3,5}.vts[2] = 6$ but $e_{2,6}.vts[3] = 4$ is smaller than $e_{3,5}.vts[3] = 5$.

Lemma V.4. ***(total order)*** *Let e_1 and e_2 be two entries. We define a strict total order '$\prec$' on the set of all entries such that $e_1 \prec e_2$ if and only if one of the following conditions holds:*

- $e_1.vts < e_2.vts$
- $e_1.vts = e_2.vts$ *and* $e_1.seq < e_2.seq$
- $e_1.vts = e_2.vts$, $e_1.seq = e_2.seq$, *and* $e_1.gid < e_2.gid$

Proof. Each group proposes entries in a sequential manner. Therefore, entries proposed by the same group are uniquely distinguished by their respective local sequence numbers seq. Since each group proposes a sequence of entries, even if two entries have the same VTS (e.g., $e_{3,4}$ and $e_{2,5}$ in Figure 6), the combination of seq and gid uniquely identifies and orders any two entries with the same VTS. □

In real-world scenarios, entries often arrive out-of-order, and different groups also receive these entries in a distinct order. Consequently, to maintain a consistent execution order, before executing an entry e, it is crucial to ensure that all entries that are ordered before e have been received and executed.

Lemma V.5. ***(monotonicity)*** *Let $e_{i,m}$ and $e_{i,n}$ be entries proposed by group G_i. If their local sequence numbers satisfy $m < n$, then it must follow that $e_{i,m} \prec e_{i,n}$.*

Proof. Each group G_j receives $e_{i,m}$ and $e_{i,n}$ and assigns its local clock time to them sequentially in order, ensuring that $e_{i,m}.vts[j] \leq e_{i,n}.vts[j]$ for any group G_j. Since $e_{i,m}.vts \leq e_{i,n}.vts$ and $e_{i,m}.seq < e_{i,n}.seq$, according to Lemma V.4, we have $e_{i,m} \prec e_{i,n}$. □

Lemma V.5 can be used to identify the next entry to be executed that is proposed by group G_i, denoted as $head_i$.

Algorithm 2: Deterministic Ordering by VTS

Input: timestamp ts_i from group G_i's clock clk_i to entry e;
Output: sequential execution of deterministically ordered entries via the `Execute` function.

1 heads: heads[i] stores the unexecuted entry proposed by group G_i with the smallest local sequence number.
2 $e.set[i]$: whether $e.vts[i]$ is inferred (*false*) or is set (*true*).
3 **Func** `OnReceiving` (ts_i, gid, seq)**:** // receive $e.vts[i]$
4 $e \leftarrow$ `GetEntry` (gid, seq); // find e by gid and seq
5 $e.vts[i] \leftarrow ts_i$; $e.set[i] \leftarrow$ *true*; // set $e.vts[i]$
6 **for** *entry* head **in** heads **do** // infer head.$vts[i]$ if not set
7 **if** head.$set[i] =$ *false* **then** head.$vts[i] \leftarrow ts_i$;
8 **while** pre $\leftarrow$ `GlobalMinimum` () **and** pre **is not** null **do**
9 `Execute` (pre);
10 nxt $\leftarrow$ `GetEntry` (pre.gid, pre.$seq + 1$);
11 heads[pre.gid] $\leftarrow$ nxt; // replacing entry pre with nxt
12 nxt.vts[nxt.gid] $\leftarrow$ nxt.seq; nxt.set[nxt.gid] $\leftarrow$ *true*;
13 **for** j **from** 0 **to** $\mathbf{n_g} - 1$ **do** // infer nxt.vts element-wise
14 **if** nxt.$set[j] =$ *false* **then** // based on pre.vts
15 nxt.$vts[j] \leftarrow$ pre.$vts[j]$;
16 **Func** `GlobalMinimum` ()**:**
17 **for** *entry* e_1 **in** heads **do**
18 **if** (**for** any other e_2 **in** heads, `Prec` (e_1, e_2) **is** *true*) **then**
19 **return** e_1; // e_1 is the most '$\prec$' unexecuted entry
20 **return** null; // other unexecuted entry e_2 may '$\prec$' than e_1
21 **Func** `Prec` (e_1, e_2)**:** // return *true* if e_1 must '$\prec$' e_2
22 **for** j **from** 0 **to** $\mathbf{n_g} - 1$ **do** // compare VTSs element-wise
23 **if** $e_1.set[j] =$ *true* **then**
24 **if** $e_1.vts[j] < e_2.vts[j]$ **then** // the increase of $e_2.vts[j]$
25 **return** *true*; // won't affect correctness
26 **if** $e_2.set[j] =$ *true* **and** $e_1.vts[j] = e_2.vts[j]$ **then**
27 **continue**; // their $vts[j]$ are identical
28 **return** *false*; // $e_2 \prec e_1$ or cannot be compared for now
29 **if** $e_1.seq \neq e_2.seq$ **then return** $e_1.seq < e_2.seq$;
30 **return** $e_1.gid < e_2.gid$; // compare vts, seq, and gid

For instance, after executing $e_{i,1}$, the next entry to be executed proposed by group G_i is $e_{i,2}$. Therefore, by comparing these next-to-execute entries proposed by each group (i.e., $heads = \langle head_1, ..., head_{\mathbf{n_g}} \rangle$), one can determine the globally next entry to be executed.

Any two entries e_1 and e_2 in $heads$ can be ordered by Lemma V.4. To reduce latency, we can establish their order before fully receiving their VTSs. This is achieved by inferring the lower bound for each element in $e_1.vts$ that has not yet been received, because each group G_i assigns its local clock time to the entries it receives in a non-decreasing order. Since timestamps are replicated via G_i's Raft instance, if another group G_j receives a timestamp $e_1.vts[i]$, any subsequent timestamp $e_2.vts[i]$ it receives cannot be less than $e_1.vts[i]$.

Algorithm 2 shows the pseudo-code of the deterministic ordering. When timestamp $e.vts[j]$ is inferred, the corresponding bit $e.set[j]$ is set to *false*. We use `Prec` to determine the order of entries. An entry e_1 orders before an entry e_2 ($e_1 \prec e_2$) if and only if `Prec` returns true. `Prec` orders entries based on the following rules:

- When $e_1.vts[j]$ is inferred, e_1 may not order before e_2, as the actual $e_1.vts[j]$ may be greater than $e_2.vts[j]$.
- Otherwise, when $e_1.vts[j]$ is set and $e_1.vts[j] < e_2.vts[j]$, the real value of $e_2.vts[j]$ must be greater than $e_1.vts[j]$ regardless of whether $e_2.vts[j]$ is inferred. As a result, e_1 orders before e_2 (lines 23-25).
- We can proceed to compare the next element only when $e_1.vts[j]$ and $e_2.vts[j]$ are equal and both have been set. Otherwise, if $e_2.vts[j]$ is inferred, its value may be greater than $e_1.vts[j]$, causing e_2 orders before e_1 (lines 26-28).
- If the VTSs of e_1 and e_2 are identical and both have been set, we use the round-based approach to determine their order by comparing their group IDs gid and sequence numbers seq (lines 29-30).

The rest of the algorithm is explained as follows: When receiving a timestamp $e.vts[i]$ of entry e, we first update e's VTS (lines 4-5) and then infer the i-th VTS of all entries in $heads$ (lines 6-7). Next, we recursively find the global next entry to be executed pre by comparing the entries in $heads$ using `Prec` (lines 16-20). If such an entry pre exists, it is executed and replaced by its successor entry nxt (lines 9-11). If any elements in the VTS of nxt have not yet been set, they can be inferred from the VTS of pre (lines 13-15).

Theorem V.6. *(*`MassBFT` *is a consensus protocol) When the network is partially synchronous, with no more than* $\mathbf{f_g}$ *groups crashed and no more than* $\mathbf{f}$ *faulty nodes per group,* `MassBFT` *satisfies the following properties:*

Agreement `MassBFT` *ensures that all correct nodes execute the same entries in the same order.*

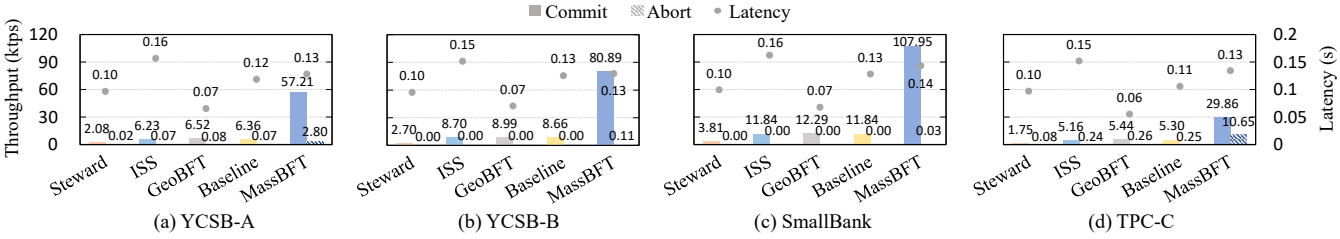
Termination `MassBFT` *ensures that each correct node eventually completes the execution of entries.*

Proof. Theorem V.3 ensures entries and their VTSs are sent atomically to all nodes. With identical VTSs as input, Lemma V.4 ensures the order of these entries is deterministic (agreement). We can infer the VTS lower bound of the next entry to be executed (Algorithm 2). As long as at least one group proposes entries, entry e's VTS will be '$\prec$' than the VTSs of other unexecuted entries, allowing e to be executed eventually (termination). □

VI. Evaluation

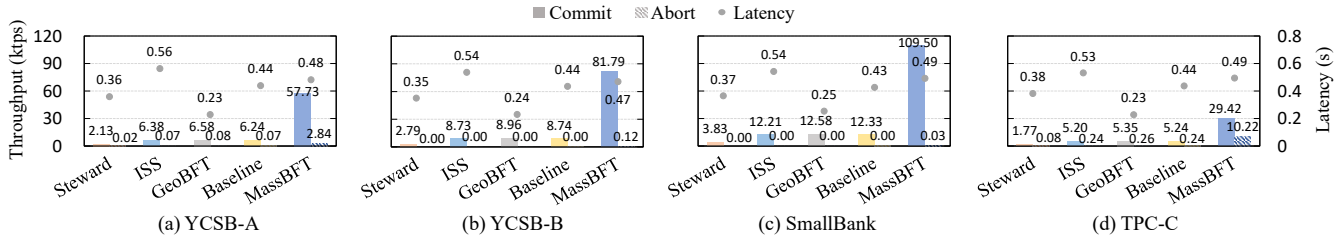
Implementation. We implement a permissioned blockchain prototype using `MassBFT` for consensus with ˜30,000 lines of C++ code. We use BFT-SMART [46] and braft [47] for local and global consensus. Since the C++ erasure code library `liberasurecode` [48] supports encoding up to a maximum of 64 chunks, which does not meet our scalability requirements, our implementation employs a high-performance Go erasure coding library [49]. We employ ED25519 to generate message digests and SHA256 to ensure data integrity. We also leverage pipelining and batching [14] [50] [51] to enhance performance.

Each group concurrently accepts local client transactions and generates a subchain of blocks. These blocks are then synchronized across groups using `MassBFT` to create a single, globally ordered, ledger. Since we measure end-to-end performance, to prevent non-consensus-related components from becoming bottlenecks, we employ Aria deterministic concurrency control [17] to accelerate transaction execution and use in-memory hash tables to store database states. Finally, clients receive the results of their transactions.



(a) YCSB-A (b) YCSB-B (c) SmallBank (d) TPC-C

Fig. 8. Performance comparison on the nationwide cluster.



(a) YCSB-A (b) YCSB-B (c) SmallBank (d) TPC-C

Fig. 9. Performance comparison on the worldwide cluster.

TABLE II
THE KEY FEATURES OF COMPETITOR SYSTEMS

System	Multi-master	Replication	Consensus	Ordering	Coding
Steward [12]	**N**	One-way	Raft	-	Entire block
ISS [53]	Y	One-way	Raft+**Epoch**	Sync.	Entire block
GeoBFT [14]	Y	One-way	**Broadcast**	Sync.	Entire block
`Baseline`	Y	One-way	Raft	Sync.	Entire block
`MassBFT` (ours)	Y	**Bijective**	Raft	**Async.**	**Erasure-coded**

Physical Environment. We deploy two geo-distributed clusters on Aliyun [52]. Each cluster consists of 3 groups, with 7 nodes in each group. For the *nationwide cluster*, the groups are located in Zhangjiakou (North China), Chengdu (West China), and Hangzhou (East China). The RTTs between any two groups range from 26.7 ms to 43.4 ms. For the *worldwide cluster*, the groups are located in Hong Kong (Asia), London (Europe), and Silicon Valley (America). The RTTs between any two groups range from 156 ms to 206 ms. Each node (an ecs.c6.2xlarge instance) is equipped with an 8-core CPU, 16 GB of memory, and runs Ubuntu 22.04 LTS. The nodes have independent network interfaces with a WAN bandwidth of 20 Mbps, while nodes within the same data center are connected through LAN with a bandwidth of 2.5 Gbps.

Competitors. For a fair comparison, we implement Steward [12], GeoBFT [14], ISS [53], and `Baseline` (Section II-A) under the same codebase with `MassBFT`. The batching timeout is fixed at 20ms for all competitors. A comparison of the key features is shown in Table II. Our implementation of Steward uses PBFT for local consensus and Raft for global consensus. GeoBFT uses PBFT for local consensus, and directly broadcasts entries to other groups. Since the Sequenced Broadcast (SB) layer of ISS is pluggable, to ensure a fair comparison, we make ISS a hierarchical consensus protocol by using Steward as its SB layer. The epoch length of ISS is 0.1s. `Baseline` employs Raft for global consensus, while other configuration parameters remain the same as in GeoBFT. We apply the optimization from GeoBFT to all other protocols. In this case, the group leader only sends the entry to $\mathbf{f_g} + 1$ nodes within each group instead of broadcasting to all nodes. Notably, all other protocols except Steward adopt the multi-master architecture with round-based synchronous ordering.

Workload. We use YCSB [54], SmallBank [55], and TPC-C [56] as our experimental workloads. In all workloads, all transactions from clients go through consensus. The YCSB workload is used to evaluate the performance under key-value workloads, using a single table with 10 columns and 1,000,000 rows, with each column having a size of 100 bytes. YCSB transactions follow a Zipf distribution with a skew factor of 0.99. We select YCSB-A (50% read and 50% write) and YCSB-B (95% read and 5% write) workloads. SmallBank workload is used to simulate bank transfer operations. We initialize 1,000,000 accounts, and the access pattern follows a uniform distribution. TPC-C workload is used to simulate order processing operations. We implement a subset of the TPC-C workload that comprises 50% NewOrder and 50% Payment transactions. We initialize 128 warehouses and execute on nodes with 64 GB of memory to accommodate these warehouses. The average transaction sizes of YCSB-A, YCSB-B, SmallBank, and TPC-C are 201B, 150B, 108B, and 232B, respectively. By default, experiments are conducted three times on the nationwide cluster under the YCSB-A workload.

A. Overall Performance

Figure 8 and Figure 9 present the performance results on the nationwide and worldwide clusters. `MassBFT` achieves the highest throughput under all workloads because its global replication strategy is more efficient than the others. ISS, GeoBFT, and `Baseline` achieve lower throughput due to the single-node bottleneck caused by replicating with group leaders. Steward has the lowest throughput because it allows only one group leader to propose entries at a time.

In Figure 8d, the throughput of `MassBFT` under TPC-C only increases by $5.64\times$ compared to `Baseline` for two main reasons. First, the transaction signature verification during local PBFT consensus consumes most computing resources, leaving insufficient resources for executing transactions, thus becoming a bottleneck in transaction processing. Second, `MassBFT` experiences a higher abort rate due to transaction conflicts. This is because the Payment transaction of TPC-C requires accessing hotspot data. As the batching timeout is

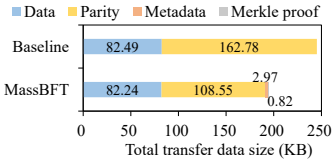


Fig. 10. WAN traffic consumption (YCSB-A).

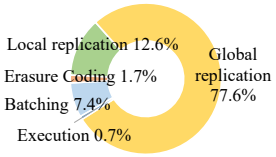


Fig. 11. Latency breakdown (YCSB-A).

fixed, MassBFT processes far more transactions per batch than Baseline (270 vs. 37), leading to a higher abort rate.

In Figure 8a, the latency of MassBFT is slightly higher than that of Baseline (i.e., 128 ms vs. 119 ms). This is because the VTS assignment (Figure 7b) introduces an extra 0.5 RTT. GeoBFT achieves the lowest latency (i.e., 68 ms) because it directly broadcasts entries to all groups without using Raft consensus during global replication, incurring only 0.5 RTT. Baseline has a higher latency than Steward due to round-based synchronization between groups. ISS has the highest latency due to its epoch-based strategy for preventing request duplication, which disrupts the Raft consensus pipeline.

Figure 9 presents the results from the worldwide cluster. All protocols show similar throughput as in nationwide experiments since pipelining effectively hides consensus latency. In Baseline and GeoBFT, the increased distance between nodes makes it difficult to keep groups synchronized, causing additional latency. Although ISS employs per-epoch synchronization to mitigate this issue, the short 0.1s epoch length results in frequent synchronizations, significantly impairing performance. To enhance throughput, we have extended its epoch length to 0.5s. The latency of MassBFT and Steward increases mainly due to the overhead of Raft consensus.

B. Replication Overhead Analysis

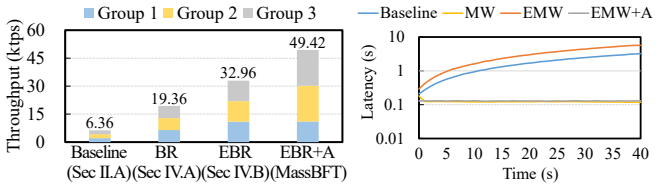
To show the replication efficiency of MassBFT, we measured the network traffic consumption when sending an entry to a remote group. As replication in MassBFT is bijective, the network overhead is the sum of the messages sent by all nodes via WAN. To ensure a fair comparison, we fixed the batch size rather than the batch timeout in this experiment, ensuring that the entry sizes for both MassBFT and Baseline are the same.

As shown in Figure 10, MassBFT consumes less WAN traffic than Baseline when replicating an entry, thanks to the utilization of erasure coding. Although MassBFT must send Merkle proofs for validating the chunks and metadata such as the PBFT certificate for validating the rebuild entry (Section IV-C), these additional costs are negligible.

Figure 11 shows a latency breakdown for MassBFT. Entry encoding and entry rebuild introduce approximately 2.3 ms of latency overhead, which is considered negligible. Most of the overhead comes from global replication, primarily due to high cross-datacenter latency. Additionally, the overhead of local consensus is also significant because nodes must verify the signatures of all transactions to prevent Byzantine faults, which is not required by global Raft consensus.

C. Results under Heterogeneous Resource

Groups with different sizes. In the nationwide cluster, we configure group G_1 with 4 nodes, while group G_2 and group G_3 have 7 nodes. We also introduce BR that only applies bijective replication (Section IV-A), EBR that applies encoded bijective replication (Section IV-B), and MassBFT with asynchronous log ordering EBR+A. Figure 12 shows the throughput breakdown and latency results.



(a) Throughput breakdown (b) Latency over time

Fig. 12. Results under different sized groups (YCSB-A).

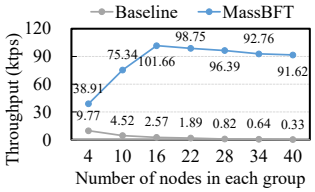
Compared to Baseline, BR achieves higher throughput due to decentralized replication. However, its lack of scalability (latency is stable because groups with different sizes have the same throughput) and impracticality in real-world scenarios (Section IV-A) limit its applicability. In EBR, throughput increases with group size but remains limited by the slowest group G_1 (note that the y-axis in Figure 12b is a log plot). MassBFT achieves the highest throughput with stable latency.

Nodes with different bandwidths. MassBFT can tolerate the presence of slow nodes within a group. As shown in Figure 14, all nodes initially have a bandwidth of 40 Mbps. We gradually increase the number of 20 Mbps nodes in each group to study the corresponding changes in throughput and latency. When the number of 20 Mbps nodes exceeds 4, the throughput decreases by 36.9%, because 5 or more slow nodes cannot be treated as crashed. According to the transfer plan (Algorithm 1), in the best case (i.e., all slow senders send their chunks to slow receivers), log replication requires only 3 correct nodes out of 7 in each group. The latency also decreases by 13.4%, as entry replication becomes the bottleneck instead of execution.

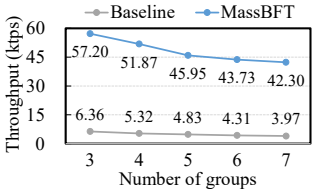
D. Scalability

Figure 13 presents the throughput results in the nationwide cluster under YCSB-A workload. To show the scalability of encoded bijective log replication, Figure 13a compares MassBFT with Baseline by scaling the number of nodes in each group from 4 to 40 (where $\mathbf{f}$ increases from 1 to 13). As the number of nodes per group increases, the throughput of Baseline decreases, as the group leader must send an entry $\mathbf{f}+1$ times to each group. Since the group leader has limited WAN bandwidth, this increased redundancy reduces the overall throughput. In contrast, MassBFT utilizes the bandwidth of all nodes to replicate entries. As the number of nodes per group increases, the aggregate bandwidth of the group also increases. Consequently, the throughput increases accordingly. When the number of nodes per group exceeds 16, throughput remains stable because transaction signature verification during local PBFT consensus becomes the bottleneck.

We also evaluate the scaling performance by increasing the number of groups. We deploy 7 nodes in each of four additional data centers (Shenzhen, Beijing, Shanghai, and Guangzhou in China) and compare MassBFT with Baseline. In Figure 13b, when the number of groups scales from 3 to



(a) Throughput when varying the number of nodes per group.



(b) Throughput when varying the number of groups.

Fig. 13. Scaling performance (YCSB-A).

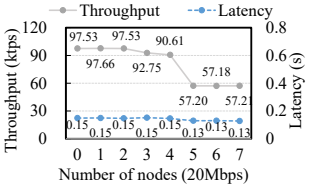


Fig. 14. Results varying node bandwidth (YCSB-A).

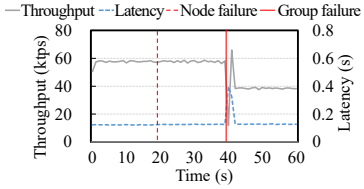


Fig. 15. Performance under failures (YCSB-A).

7, the throughput of MassBFT decreases from 57.20 ktps to 42.30 ktps, representing a decrease of **26.0**%. In contrast, the throughput of Baseline decreases from 6.36 ktps to 3.97 ktps, representing a decrease of **37.6**%. This is because the global Raft consensus is not scalable [57] [58], and increasing the number of groups leads to higher replication overhead.

E. Fault Tolerance

To simulate node failures, each group contains two Byzantine nodes, and Byzantine nodes of all groups collude to replicate a tampered entry. To simulate data center crashes, we simultaneously kill all nodes in a group. Figure 15 shows the throughput and latency results in the nationwide cluster.

Node Failures. All Byzantine nodes always strictly follow the local consensus process and possess the same tampered entries. From the 20th second (the red dotted line), the Byzantine sending nodes encode these tampered entries (instead of the correct ones) into chunks and begins the encoded bijective log replication. Meanwhile, the Byzantine receiving nodes broadcast the chunks encoded from the tampered entries within its group. Additionally, all chunks sent by Byzantine nodes are received by the correct nodes. Figure 15 shows that the throughput of MassBFT remains unchanged because the correct nodes refuse to receive from the Byzantine nodes after rebuilding the tampered entries. As a result, a correct node can only receive chunks from other correct nodes to rebuild the entry, causing about a 3 ms increase in latency.

Group Failures. In Figure 15, group G_0 crashes at the 40th second (the red solid line). After it crashes, G_0 cannot assign timestamps from its clock clk_0 to the newly replicated entries. Since deterministic ordering requires the timestamp $e.vts[0]$ of entry e to be set before execution (Algorithm 2), entries can only continue replication but cannot be ordered or executed. As a result, latency increases and throughput reduces. After a timeout occurs [12] [13], a new leader is elected and starts assigning timestamps from G_0's clock clk_0 to entries, resulting in a reduction in latency and an increase in throughput. Finally, the throughput remains lower because the crashed group cannot propose entries.

VII. Related work

Asynchronous Consensus. Asynchronous consensus protocols [59]–[68] are leaderless and are designed to operate in asynchronous networks [69]. HBBFT [59] and Dumbo [60], [61] employ ABA [70] and MVBA [71] [72], respectively, to identify and exclude the entries proposed by slow nodes. [63] proposes the Mempool protocol Narwhal for entry replication and the asynchronous consensus Tusk for entry ordering. Narwhal stores entries in a DAG structure to maintain their causal order, enabling Tusk to establish a total order efficiently. Some protocols [66]–[68], adopt an optimistic fast lane to bypass ABA and MVBA. However, the fast lane is hard to achieve in practice [64]. Dumbo-NG [64] accelerates consensus by decoupling entry broadcast and ordering and leveraging pipelines, which is close to our work. However, it still suffers from high MVBA latency, which pipelining cannot hide. In contrast, MassBFT adopts Raft for global replication, which offers lower latency and tolerates $\lfloor(\mathbf{n_g}-1)/2\rfloor$ instead of $\lfloor(\mathbf{n_g}-1)/3\rfloor$ group failures since groups are crash-only.

Erasure Coding. In the CFT context, RS-Paxos [73] uses erasure coding [35] to reduce storage and network costs, but has limitations on the number of faulty nodes it can tolerate. [74]–[77] adopt different encoding and transmission strategies depending on the number of correct nodes available. In the BFT context, [78]–[82] use carefully designed encoding schemes to alleviate the leader bandwidth bottleneck. [40] proposes the delayed-replication algorithm to reduce the communication cost of notifying learners. RapidChain [83] and [84] integrate gossip with erasure codes to improve liveness in large networks. HBBFT [59] and Dumbo [60] integrate erasure codes into the reliable broadcast protocol [85], reducing network traffic during replication. BEAT [86] and ABFT [87] use optimized erasure codes to save bandwidth further. In addition, [88]–[94] apply erasure codes to reduce costs for BFT storage.

These protocols adopt a flattened architecture without fully utilizing the heterogeneous networks within and between data centers. In contrast, MassBFT adopts a hierarchical architecture that integrates erasure coding into group sending, eliminating leader bottlenecks while significantly reducing WAN traffic.

VIII. Conclusion

This paper introduces MassBFT, a geo-distributed BFT consensus protocol designed for high performance and scalability. We propose encoded bijective log replication to reduce transmission redundancy, and asynchronous log ordering to enable groups to propose entries at their own pace without synchronizations. Experiments show that MassBFT outperforms SOTA protocols by a factor ranging from 5.49 to 29.96.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (62372097, U2241212, and 62072082), the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501), and the Fundamental Research Funds for the Central Universities (N2416003). Yanfeng Zhang is the corresponding author.

REFERENCES

[1] S. Gupta, S. Rahnama, E. Linsenmayer, F. Nawab, and M. Sadoghi, "Reliable transactions in serverless-edge architecture," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 301–314.

[2] M. J. Amiri, Z. Lai, L. Patel, B. T. Loo, E. Lo, and W. Zhou, "Saguaro: An edge computing-enabled hierarchical permissioned blockchain," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 259–272.

[3] Y. Peng, M. Du, F. Li, R. Cheng, and D. Song, "FalconDB: Blockchain-based collaborative database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 637–652. [Online]. Available: https://doi.org/10.1145/3318464.3380594

[4] Z. Peng, J. Xu, H. Hu, L. Chen, and H. Kong, "BlockShare: A blockchain empowered system for privacy-preserving verifiable data sharing," *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng*, vol. 1, pp. 14–24, 2022.

[5] R. Neiheiser, L. Rech, M. Bravo, L. Rodrigues, and M. Correia, "Fireplug: Efficient and robust geo-replication of graph databases," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1942–1953, 2020.

[6] Z. Hong, S. Guo, E. Zhou, W. Chen, H. Huang, and A. Zomaya, "GriDB: Scaling blockchain database via sharding and off-chain cross-shard mechanism," *Proc. VLDB Endow.*, vol. 16, no. 7, pp. 1685–1698, 2023.

[7] S. Hamdan, M. Ayyash, and S. Almajali, "Edge-computing architectures for internet of things applications: A survey," *Sensors*, vol. 20, no. 22, p. 6441, 2020.

[8] D. Loghin, S. Cai, G. Chen, T. T. A. Dinh, F. Fan, Q. Lin, J. Ng, B. C. Ooi, X. Sun, Q.-T. Ta *et al.*, "The disruptions of 5g on data-driven technologies and applications," *IEEE transactions on knowledge and data engineering*, vol. 32, no. 6, pp. 1179–1198, 2020.

[9] Z. Yang, Q. Zhou, L. Lei, K. Zheng, and W. Xiang, "An iot-cloud based wearable ecg monitoring system for smart healthcare," *Journal of medical systems*, vol. 40, pp. 1–11, 2016.

[10] N. Chen, Y. Chen, Y. You, H. Ling, P. Liang, and R. Zimmermann, "Dynamic urban surveillance video stream processing using fog computing," in *2016 IEEE second international conference on multimedia big data (BigMM)*. IEEE, 2016, pp. 105–112.

[11] W. Saad, M. Bennis, and M. Chen, "A vision of 6g wireless systems: Applications, trends, technologies, and open research problems," *IEEE network*, vol. 34, no. 3, pp. 134–142, 2019.

[12] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling byzantine fault-tolerant replication to wide area networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 80–93, 2010.

[13] F. Nawab and M. Sadoghi, "Blockplane: A global-scale byzantizing middleware," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 124–135.

[14] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "ResilientDB: Global scale resilient blockchain fabric," *Proc. VLDB Endow.*, vol. 13, no. 6, p. 868–883, feb 2020. [Online]. Available: https://doi.org/10.14778/3380750.3380757

[15] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OsDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[16] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, p. 398–461, nov 2002. [Online]. Available: https://doi.org/10.1145/571637.571640

[17] Y. Lu, X. Yu, L. Cao, and S. Madden, "Aria: A fast and practical deterministic oltp database," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2047–2060, jul 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407808

[18] F. Liu and Y. Yang, "D-Paxos: Building hierarchical replicated state machine for cloud environments," *IEICE Transactions on Information and Systems*, vol. E99.D, no. 6, pp. 1485–1501, 2016.

[19] T. Castiglia, C. Goldberg, and S. Patterson, "A hierarchical model for fast distributed consensus in dynamic networks," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 1189–1190.

[20] M. J. Amiri, D. Shu, S. Maiyya, D. Agrawal, and A. El Abbadi, "Ziziphus: Scalable data management across byzantine edge servers," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 490–502.

[21] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[22] S. Keshav, W. Golab, B. Wong, S. Rizvi, and S. Gorbunov, "RCanopus: Making canopus resilient to failures and byzantine faults," 2019.

[23] J. Hellings and M. Sadoghi, "Brief announcement: The fault-tolerant cluster-sending problem," in *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2019.

[24] J. Hellings and M. Sadoghi, "The fault-tolerant cluster-sending problem," in *International Symposium on Foundations of Information and Knowledge Systems*. Springer, 2022, pp. 168–186.

[25] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[26] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 279–296.

[27] R. Neiheiser, M. Matos, and L. Rodrigues, "Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 35–48. [Online]. Available: https://doi.org/10.1145/3477132.3483584

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

[29] S. Rizvi, B. Wong, and S. Keshav, "Canopus: A scalable and massively parallel consensus protocol," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 426–438. [Online]. Available: https://doi.org/10.1145/3143361.3143394

[30] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319.

[31] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "CockroachDB: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1493–1509. [Online]. Available: https://doi.org/10.1145/3318464.3386134

[32] J. Kończak, N. F. de Sousa Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper, "JPaxos: State machine replication based on the paxos protocol," 2011.

[33] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–356. [Online]. Available: https://doi.org/10.1145/3293611.3331591

[34] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, p. 288–323, apr 1988. [Online]. Available: https://doi.org/10.1145/42282.42283

[35] R. E. Blahut, "Theory and practice of error control codes," 1983.

[36] Y. Wang, S. Jain, M. Martonosi, and K. Fall, "Erasure-coding based routing for opportunistic networks," in *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, 2005, pp. 229–236.

[37] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 2, p. 24–36, apr 1997. [Online]. Available: https://doi.org/10.1145/263876.263881

[38] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern, "Scalable secure storage when half the system is faulty," in *International Collo-*

quium on Automata, Languages, and Programming. Springer, 2000, pp. 576–587.

[39] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern, "Addendum to scalable secure storage when half the system is faulty," *Information and Computation*, 2004.

[40] J. Hellings and M. Sadoghi, "Coordination-free byzantine replication with minimal communication costs," in *23rd International Conference on Database Theory (ICDT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[41] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.

[42] L. Ramabaja and A. Avdullahu, "Compact merkle multiproofs," *arXiv preprint arXiv:2002.07648*, 2020.

[43] B. Arun and B. Ravindran, "Scalable byzantine fault tolerance via partial decentralization," *Proc. VLDB Endow.*, vol. 15, no. 9, p. 1739–1752, may 2022. [Online]. Available: https://doi.org/10.14778/3538598.3538599

[44] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE transactions on dependable and secure computing*, vol. 8, no. 4, pp. 564–577, 2010.

[45] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*. New York, NY, USA: Association for Computing Machinery, 2019, p. 179–196. [Online]. Available: https://doi.org/10.1145/3335772.3335934

[46] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 355–362.

[47] "braft: An industrial-grade c++ implementation of raft consensus algorithm," 2023. [Online]. Available: https://github.com/baidu/braft

[48] "liberasurecode," 2024. [Online]. Available: https://github.com/openstack/liberasurecode

[49] "Reed-solomon erasure coding in go," 2023. [Online]. Available: https://github.com/klauspost/reedsolomon

[50] J. Qi, X. Chen, Y. Jiang, J. Jiang, T. Shen, S. Zhao, S. Wang, G. Zhang, L. Chen, M. H. Au *et al.*, "Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 18–34.

[51] Z. Peng, Y. Zhang, Q. Xu, H. Liu, Y. Gao, X. Li, and G. Yu, "NeuChain: A fast permissioned blockchain system with deterministic ordering," *Proc. VLDB Endow.*, vol. 15, no. 11, p. 2585–2598, jul 2022. [Online]. Available: https://doi.org/10.14778/3551793.3551816

[52] "Alibaba Cloud: Cloud computing services," 2023. [Online]. Available: https://www.alibabacloud.com/

[53] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, "State machine replication scalability made simple," in *Proceedings of the Seventeenth EuroSys Conference*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 17–33. [Online]. Available: https://doi.org/10.1145/3492321.3519579

[54] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[55] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," *ACM Trans. Database Syst.*, vol. 34, no. 4, dec 2009. [Online]. Available: https://doi.org/10.1145/1620585.1620587

[56] "Tpc benchmark c," 2023. [Online]. Available: http://www.tpc.org/tpcc/

[57] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 358–372. [Online]. Available: https://doi.org/10.1145/2517349.2517350

[58] M. Kogias and E. Bugnion, "HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–17.

[59] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 31–42. [Online]. Available: https://doi.org/10.1145/2976749.2978399

[60] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 803–818.

[61] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *Proceedings of the 39th symposium on principles of distributed computing*, 2020, pp. 129–138.

[62] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," *CoRR*, vol. abs/2102.08325, 2021. [Online]. Available: https://arxiv.org/abs/2102.08325

[63] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and Tusk: a dag-based mempool and efficient bft consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 34–50.

[64] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1187–1201.

[65] N. Giridharan, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Bullshark: DAG BFT protocols made practical," *CoRR*, vol. abs/2201.05677, 2022. [Online]. Available: https://arxiv.org/abs/2201.05677

[66] Y. Lu, Z. Lu, and Q. Tang, "Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2159–2173.

[67] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *International conference on financial cryptography and data security*. Springer, 2022, pp. 296–315.

[68] S. Liu, W. Xu, C. Shan, X. Yan, T. Xu, B. Wang, L. Fan, F. Deng, Y. Yan, and H. Zhang, "Flexible advancement in asynchronous bft consensus," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 264–280. [Online]. Available: https://doi.org/10.1145/3600006.3613164

[69] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[70] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous byzantine consensus with t_i n/3 and o (n2) messages," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, 2014, pp. 2–9.

[71] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Annual International Cryptology Conference*. Springer, 2001, pp. 524–541.

[72] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous byzantine agreement," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 337–346.

[73] S. Mu, K. Chen, Y. Wu, and W. Zheng, "When paxos meets erasure code: Reduce network and storage cost in state machine replication," in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 61–72. [Online]. Available: https://doi.org/10.1145/2600212.2600218

[74] Z. Wang, T. Li, H. Wang, A. Shao, Y. Bai, S. Cai, Z. Xu, and D. Wang, "CRaft: An erasure-coding-supported version of raft for reducing storage cost and network cost," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 297–308.

[75] Y. Jia, G. Xu, C. W. Sung, S. Mostafa, and Y. Wu, "HRaft: Adaptive erasure coded data maintenance for consensus in distributed networks," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 1316–1326.

[76] M. Xu, Y. Zhou, Y. Y. Qiao, K. Xu, Y. Wang, and J. Yang, "ECRaft: A raft based consensus protocol for highly available and reliable erasure-coded storage systems," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, 2021, pp. 707–714.

[77] M. Zhang, Q. Kang, and P. P. C. Lee, "Minimizing network and storage costs for consensus with flexible erasure coding," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 41–50. [Online]. Available: https://doi.org/10.1145/3605573.3605619

[78] N. Chawla, H. W. Behrens, D. Tapp, D. Boscovic, and K. S. Candan, "Velocity: Scalability improvements in block propagation through rateless erasure coding," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2019, pp. 447–454.

[79] J. Camenisch, M. Drijvers, T. Hanke, Y.-A. Pignolet, V. Shoup, and D. Williams, "Internet computer consensus," in *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, 2022, pp. 81–91.

[80] I. Kaklamanis, L. Yang, and M. Alizadeh, "Poster: Coded broadcast for scalable leader-based bft consensus," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 3375–3377. [Online]. Available: https://doi.org/10.1145/3548606.3563494

[81] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "DispersedLedger: High-throughput byzantine consensus on variable bandwidth networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 493–512.

[82] X. Li, L. Cai, W. Qiu, F. Huang, and Z. Lei, "Segmenta: Pipelined bft consensus with slicing broadcast," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2023, pp. 101–120.

[83] M. Zamani, M. Movahedi, and M. Raykova, "RapidChain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 931–948. [Online]. Available: https://doi.org/10.1145/3243734.3243853

[84] C. Santiago and C. Lee, "Accelerating message propagation in blockchain networks," in *2020 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2020, pp. 157–160.

[85] G. Bracha, "An asynchronous [(n-1)/3]-resilient consensus protocol," in *Proceedings of the third annual ACM symposium on Principles of distributed computing*, 1984, pp. 154–162.

[86] S. Duan, M. K. Reiter, and H. Zhang, "BEAT: Asynchronous bft made practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2028–2041.

[87] J. Wang, J. Ou, M. Zhou, X. Wu, Y. Luo, H. Hu, and Y. Xiao, "ABFT: high-performance asynchronous byzantine fault-tolerant consensus algorithm for electricity data metrology," in *Second International Conference on Energy, Power, and Electrical Technology (ICEPET 2023)*, vol. 12788. SPIE, 2023, pp. 1137–1146.

[88] E. Androulaki, C. Cachin, D. Dobre, and M. Vukolić, "Erasure-coded byzantine storage with separate metadata," in *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*. Springer, 2014, pp. 76–90.

[89] X. Qi, Z. Zhang, C. Jin, and A. Zhou, "A reliable storage partition for permissioned blockchain," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 1, pp. 14–27, 2020.

[90] H. Guo, M. Xu, J. Zhang, C. Liu, R. Ranjan, D. Yu, and X. Cheng, "BFT-DSN: A byzantine fault tolerant decentralized storage network," *IEEE Transactions on Computers*, 2024.

[91] X. Qi, Z. Zhang, C. Jin, and A. Zhou, "BFT-Store: Storage partition for permissioned blockchain via erasure coding," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1926–1929.

[92] D. Xiang, R. Zhou, L. Ma, Q. Zeng, X. Fu, Q. Wang, B. Chen, and Y. Jia, "Decoupling consensus and storage in consortium blockchains by erasure codes," in *2022 4th International Conference on Data Intelligence and Security (ICDIS)*. IEEE, 2022, pp. 194–199.

[93] H. Wu, A. Ashikhmin, X. Wang, C. Li, S. Yang, and L. Zhang, "Distributed error correction coding scheme for low storage blockchain systems," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7054–7071, 2020.

[94] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter, "Efficient byzantine-tolerant erasure-coded storage," in *International Conference on Dependable Systems and Networks, 2004*. IEEE, 2004, pp. 135–144.