EasyChair Preprint

№ 1713

LgTTBFT : Effective Byzantine Fault Tolerance Algorithm Based on Structured Network and Trusted Execution Environment

Rihong Wang, Na Li, Quanqing Xu, Lifeng Zhang and Congying Xing

October 18, 2019

LgTTBFT：Effective Byzantine Fault Tolerance Algorithm Based on Structured Network and Trusted Execution Environment

RiHong Wang[1] , Na Li[1(✉)], Quanqing Xu[2], LiFeng Zhang[1] and Congying Xing[1]

[1] Qingdao University of Technology, Qingdao Shandong 266520, China

congyinglee121@gmail.com

[2]Ant Financial, Hangzhou Zhejiang

Abstract. The BFT consensus algorithm provides 100% security rather than probability, but it has not been widely used due to its high resource consumption and low consensus efficiency. The blockchain community has also been optimized on the basis of traditional Byzantine fault tolerance, trying to actually apply the BFT protocol to the consensus layer of the blockchain.

In this paper, we propose *Loop-Grouping Tree and TEE BFT* (LgTTBFT), a simple and efficient Byzantine fault tolerant consensus strategy. At the heart of LgTTBFT is a new tree topology (LgTree) and Trusted Execution Environment (TEE). We constructed a structured network and multi-reconstructed multi-way tree structure LgTree as the node organization structure of the network layer of the blockchain system. Compared with MinBFT and Ethereum, it provides a simpler network structure and more efficient routing efficiency. The special tree structure also performs well in terms of load balancing and stability. The use of TEE has greatly improved system efficiency from both the number of replicas and the communication process: reduce the minimum number of replicas from $3f+1$ to $2f+1$, The communication phase of the consensus process is reduced from three phases of PBFT to two phases, and the communication complexity of the PREPARE phase is reduced from $O(n^2)$ to $O(n)$. Experiments have shown that the combination of structured network and trusted execution environment provides better throughput and latency.

Keywords: Blockchain, Byzantine fault-tolerance, structured network, trusted components, state machine replication

1. Introduction

The surging interest in blockchain technology has revived the exploration of the Byzantine fault tolerance consensus. The bottleneck of blockchain is mainly concentrated on its performance problems of low throughput and high latency, which is actually the result of the trade-off between decentralization and efficiency. Classic distributed consensus algorithms such as Paxos [1] and Raft [2] cannot solve the Byzantine fault tolerance problem, however, the consensus algorithms of public chain systems such as PoW [3] and PoS [4] cannot meet the performance requirements in specific scenarios due to their huge energy costs or design defects of the protocol itself. In contrast, the BFT (Byzantine Fault Tolerant) consensus algorithm can tolerate a certain number of nodes to deviate from the protocol in any way. Byzantine error is the worst-case error in distributed systems.

The blockchain network is a decentralized P2P (Peer-to-Peer) network. The characteristics of no central node give the blockchain system reliable security. The shortcomings of this topology are also obvious: distributed node deployment makes communication between nodes very inefficient. This led to two classes exploring in the future work: unstructured P2P networks and structured P2P networks. The former is inefficient and unstable, while the latter is efficient in searching but complex in structure [5,6,7].

The P2P network of the Bitcoin main network is unstructured, but the Ethereum's P2P network is structured. The underlying distributed network of Ethereum uses the classic Kademlia [8] network. The core data structure of the network consists of a routing table called K-bucket, and the entire network topology is a binary trie. K-bucket is sorted according to the logical distance from the target node, a total of 256 K-buckets, and each K-bucket contains 16 nodes. KRaft [39] is a Raft algorithm, and the process of Leader election and log replication in the Raft algorithm was optimized by using Kademlia protocol.

Experimental results showed that TPS increased by 41%. The Kad communication protocol in Ethereum is a UDP-based P2P node discovery protocol, which can quickly and accurately route and locate data in a distributed environment, but the network structure is too complicated.

Trusted Execution Environment (TEE) is a technology that protects data and programs in an isolated manner, providing a secure area that ensures the security, confidentiality, and integrity of the code and data loaded into the environment. The Intel SGX [9] (Software Guard Extensions) module provides support for TEE. Combined with blockchain technology, it can solve blockchain data privacy problems and support for complex calculations without affecting the decentralization of blockchain and the inflexibility of data. Applying TEE at the consensus level can reduce the number of replicas and the communication phase of the BFT protocol by setting trusted counters.

In this paper, we present LgTTBFT, a simple and efficient Byzantine fault-tolerant consensus strategy. The core of LgTTBFT is a multi-way tree structure based on structured network, which we called LgTree. Its unique tree topology provides low storage overhead and high efficiency, while achieving good load balancing. In addition, the introduction of TEE reduces the number of replicas in the network to $2f+1$, while reducing the communication overhead of the PREPARE phase to linear.

Our experiments show that, compared to the MinBFT we evaluated also using the Trusted Execution Environment but not adding a structured network, the throughput of LgTTBFT is at least twice that of MinBFT, and the performance advantage is more obvious as the number of replicas increases.

In summary, we make the following contributions:

- The LgTree structure is proposed and its construction algorithm is implemented. The tree structure is used as the node organization structure of the network layer of the blockchain, and a simple and efficient BFT consensus mechanism is realized.
- A node routing algorithm based on LgTree is proposed to achieve efficient search efficiency.
- The LgTTBFT and related BFT systems were evaluated and the results showed that it is superior to BFT variants such as MinBFT and Ethereum in efficiency (throughput and delay).

2. Preliminaries

In this section, we describe the known BFT protocol and existing optimizations, and provide a brief overview of the key technologies used in the LgTTBFT protocol.

2.1 BFT Consensus Optimization

Byzantine Generals Problem and its initial solution were proposed by Lamport in 1982 [10], The OM(m) and SM(m) algorithms, in addition to requiring numerous assumptions, the exponential algorithm efficiency also makes it stay Interest in theory. The emergence of PBFT (Practical Byzantine Fault Tolerance) [11] in 1999 reduced the complexity of the algorithm from exponential to polynomial, which really made the Byzantine fault-tolerant algorithm feasible in practical system applications. Even so, applying PBFT to a blockchain scenario can not meet its performance requirements.

Many research efforts in recent years have helped to make the BFT system more feasible [12, 13, 14]. It mainly includes improvements in these two aspects: 1) In terms of **algorithm complexity**, MinBFT [15] simplifies the traditional BFT consensus communication round through the introduction of Trusted Execution Environment (TEE), and the communication cost reduction achieves higher throughput than PBFT; CheapBFT [16] implements an *optimistic BFT* strategy while introducing a trusted execution environment, requiring only $f+1$ active replicas to participate, and normal use of passive replication to save resources, aiming to establish a resource-saving BFT system; Both MinBFT and CheapBFT have improved the communication efficiency of the PREPARE phase in PBFT, and the introduction of FastBFT [17] has brought the performance of the BFT consensus algorithm to a new level. Its core is a novel message aggregation technology that combines the trusted execution environment with lightweight secret sharing. It optimizes the communication efficiency of the COMMIT phase for the first time and reduces the algorithm complexity to $O(n)$; 2) In the **fault-tolerant rate**, the introduction of the trusted execution environment reduces the fault tolerance of the BFT consensus algorithm from $3f+1$ to $2f+1$ (that is, only $2f+1$ nodes can tolerate f fault nodes); With the Optimistic BFT strategy, only $f+1$ active replicas need to participate in the consensus in normal case.

From a specific application scenario, the BFT consensus algorithm is optimized for the public chain

mainly from the following two aspects: 1) Combined with Nakamoto consensus (including NEO's DBFT (Delegated Byzantine Fault Tolerant) [18], Stellar's FBA (Federated Byzantine Agreement) [19], Tendermint BFT [20], etc.); 2) Cryptographic-based optimization (including ByzCoin [21], Zilliqa [22] using aggregated signatures, Algorand [23] using verifiable random functions, VBFT [24], and HoneyBadgerBFT [25] using threshold signatures. These BFT consensus algorithms are applicable to specific application scenarios, with different degrees of trade-offs between decentralization and algorithm efficiency.

These consensus mechanisms use different technologies and strategies to improve the BFT algorithm to varying degrees, so that the BFT consensus can be truly applied to the production environment. However, for a system as a whole, the improvement of the consensus layer may be limited, and the improvement with other layers can achieve better efficiency. As far as we know, there is no consensus algorithm to optimize the node organization structure of P2P network. Therefore, we propose effective Byzantine fault tolerance based on structured network LgTree and Trusted Execution Environment (TEE).

2.2 Structured P2P network

Unstructured P2P networks, such as the fully distributed Gnutella [26] network, need to traverse the entire P2P network when performing search requests, and the bandwidth consumption increases exponentially with the increase in the number of users. In addition, nodes may leave the network frequently, so the Gnutella network structure is very unstable and inefficient, and cannot be applied to the actual environment.

The structured P2P network is established on the logical network, for example, forming a ring network or a tree network, and the route query algorithm can be designed according to the logical structure, thereby avoiding flooding-search and obtaining high communication efficiency between nodes. Distributed Hash Table (DHT) is a typical implementation of structured networks [27]. The core idea is that each client is responsible for a small range of routes and is responsible for storing a small amount of data, thereby addressing and storage of the entire DHT network is achieved. Chord [28], Pastry [29] and Kademlia are typical DHT implementation algorithms.

In this paper, we have carried out a similar design to Ethereum, and proposed a network node organization model and corresponding routing algorithm. The design of this structure is more concise and efficient than Ethereum.

2.3 Trusted Execution Environment (TEE)

Currently, hardware security mechanisms have been widely used on commercial computing platforms. Intel SGX provides a trusted execution environment called "*Enclave*" that prevents other applications, including operating systems, BIOS systems, from snooping and tampering with the state and data of protected applications.

Byzantine algorithms can tolerate nodes deviating from the protocol in any way. In a protocol based on the state machine replication model, the implementation of trusted counters can provide a trusted sequence of operations so that malicious replicas cannot allow different correct replicas to perform different operations. Compared to other trusted services (such as A2M [30], TrInc [31], EBAWA [32], ZZ [33]), Intel SGX is currently the simplest tamper-proof component. In this paper, we implemented a trusted counter using SGX.

First, the trusted monotonic counter generates an unforgeable counter value and can only use some kind of cryptographic primitive-based authentication. Second, the counter binds the sequence numbers to the operation, which assigns a unique serial number to the client request, which guarantees that the same identifier will never be assigned to two different messages. In addition, the tamper-proof component generates a signature certificate that demonstrates the binding of Numbers to messages and the incrementation of counters.

3. LgTTBFT Overview

Both MinBFT and CheapBFT introduce a trusted execution environment, but there is no improvement to the consensus model at the network layer. The MinBFT algorithm only introduces a trusted execution environment, but does not use a structured network. Each node must establish a connection with all nodes when it starts up, which is very inefficient and wasteful resources in P2P networks. As the number of

nodes increases, the load on each node will also increase greatly.

CheapBFT is actually a Speculation-based Byzantine protocol that uses a more optimistic strategy. The assumption of this strategy is that the server is in a normal state most of the time. Under normal circumstances, only $f+1$ active replicas participate in the protocol, and the other replicas are passively copied. However, when a Byzantine server exists in the system, system performance will drop dramatically.

Since FastBFT has proposed the use of secret sharing techniques to optimize PBFT, we have not applied secret sharing techniques to this consensus algorithm in this paper. Perhaps the combination of structured network and secret sharing technology can better optimize the BFT algorithm, which is worthy of further study.

Although the introduction of TEE into BFT has been proposed by many previous work (MinBFT, CheapBFT, FastBFT), as far as we know, the combination of TEE and network layer improvement (structured P2P network) has not been explored by the research team. In this paper, we have combined with TEE based on the proposed efficient structured network, and have achieved high performance through experiments.

3.1 System Model

Our BFT system is based on the traditional state machine replication model. Unlike PBFT, the introduction of TEE requires only $2f+1$ nodes to tolerate f faulty nodes. It operates in the same environment as PBFT and requires weak synchronization to maintain liveness. We assume that each replica has a hardware-based TEE that is used to maintain a monotonic counter. In addition, we designed and implemented a structured network layer node distribution, enabling the entire system to achieve high routing efficiency and load balancing while having a simple architecture.

The network situation of a distributed system is complex, and the network may cause loss, delay, reordering, or misdelivery of messages that are passed between the replica and the client. For simplicity, we used an abstraction of the FIFO channel when implementing the protocol.

In this paper, we only discuss the architecture and efficiency of the system in normal case, and we will not consider the situation of view-change. But view-change is a very important part of the BFT consensus, which deserves our continued research in future work.

3.2 Normal-case Operation

LgTTBFT follows a message exchange pattern similar to PBFT (see Fig. 1), but provides a simpler communication phase and communication complexity. Figure 1 depicts the detailed differences between the messages passed by LgTTBFT and PBFT during the consensus process. For the message mode, the message pattern generated by simply using TEE to improve the communication process is about the same. This message pattern is similar to MinBFT and EBFT. Here we mainly describe the simplification of the Prepare phase brought by TEE. See Section 6 for the performance gains associated with the combination of TEE and structured networking.

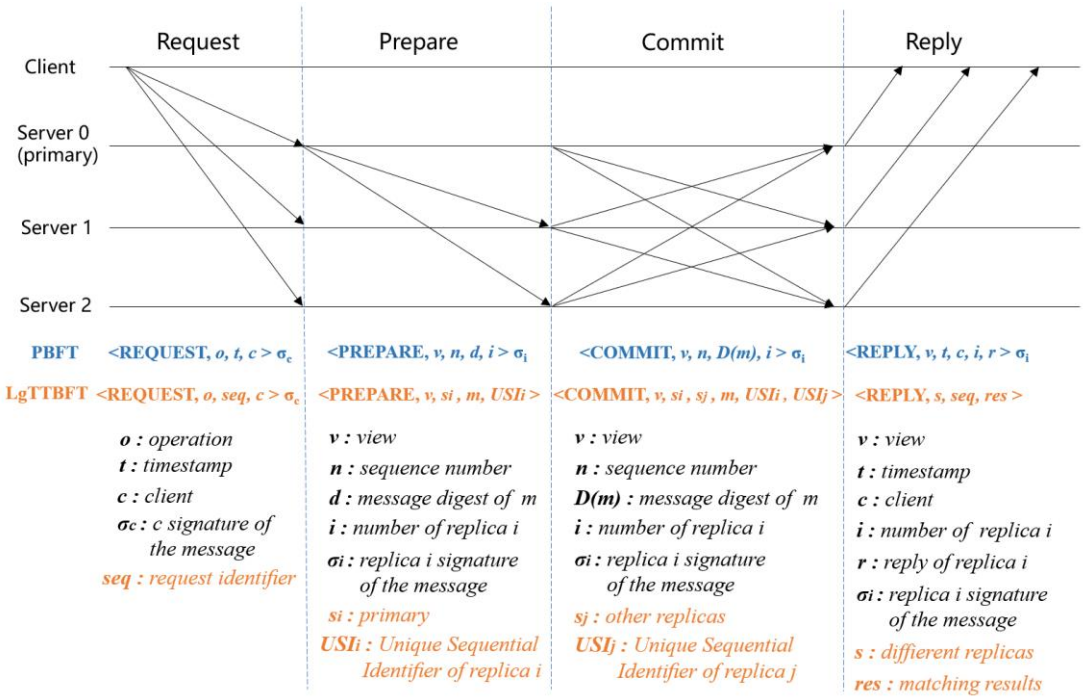

Fig. 1. Message patterns of LgTTBFT.

The sequence of events in normal case is as follows:

Request. Client ***c*** requests the execution of operation ***o*** by sending a signature message <REQUEST, *o, t, c* > σc to the *primary*. Here ***seq*** is a request identifier used to ensure one-time semantics, requesting to sign using the client's private key. After sending the request, the client waits for a reply from $f+1$ different replicas ***s***. Since the maximum number of failure replicas is f, this ensures that at least one reply is from the non-failure replica.

Prepare. After receiving the request from the client, the primary uses the trusted counter to obtain a Unique Sequence Identifier (***USIi***) and binds the value to the request. The primary s_i then multicasts the request to all replicas via the <PREPARE, v, s_i, m, USI_i > message.

Commit. After receiving the PREPARE message, each replica s_j will perform a series of verifications on the content of the message. After the verification succeeds, the message ***m*** is resent to all other nodes through the <COMMIT, $v, s_i, s_j, m, USI_i, USI_j$ > messages. Here, each COMMIT message carries the sender's unique sequence identifier USI_j, so no two messages have the same identifier, and the replica can check that the identifier they received is valid for the message.

Reply. After received $f+1$ correct COMMIT messages and verified, the replica performs operations to obtain the result ***res*** and send a REPLY message to the client.

It can be seen that the LgTTBFT algorithm has one communication step less than the PBFT, and the communication complexity of the PREPARE phase is $O(n)$ instead of $O(n^2)$.

4. LgTree Structured Network

The LgTree (Loop-Grouping Tree) structured network we designed is aimed at build a simple and efficient network node organization structure, and provide high routing efficiency and load balancing by designing corresponding routing algorithm.

4.1 LgTree Structure

We organize all the nodes in the network into a tree structure. The constructor only needs to calculate the number of groups ***g*** and the number of elements ***e*** of each group as input to construct the tree structure. The calculation methods for ***g*** and ***e*** are given in Algorithm 1:

Algorithm 1. Compute ***g*** and ***e***

Input: Total number of nodes, ***n***

Output: Number of groups, ***g*** ; Number of elements in each group, ***e***

```
sqrtValue ⟵ sqrt (n)
sqrtValueInt ⟵ int (sqrtValue)
if sqrtValue ≥ sqrtValueInt +1/2 then
    g ⟵ ceil (sqrtValue)
    e ⟵ ceil (sqrtValue)
    return g , e
else
    g ⟵ ceil (sqrtValue)
    e ⟵ int (sqrtValue)
    return g , e
```

Compute *g* and *e*. Let the total number of nodes in the network be ***n***. According to the design of the following virtual matrix, we assume $g \times e \geq n$. Therefore, first take the square root of ***n*** to get the result *sqrtValue*, if the fractional part of *sqrtValue* is less than 1/2, then

$$g = \lceil \sqrt{n} \rceil \quad ; \quad e = \lceil \sqrt{n} \rceil \tag{1}$$

the values of ***g*** and ***e*** are the result of rounding up and rounding, respectively. If the fractional part of *sqrtValue* is greater than or equal to 1/2, then ***e*** must also be rounded up, otherwise $g \times e < n$, ie

$$g = e = \lceil \sqrt{n} \rceil \tag{2}$$

Node ID. The ID of each node consists of two parts: ***gID*** and ***eID***. Where ***gID*** indicates that the node is located in which group, and ***eID*** indicates that the node is which element in the group. For example, if there are 10,000 nodes, then $g = e = 100$ and the ID of the first node is 001001. As shown in Fig. 2, we use different colors to represent different ***eIDs***, group all nodes and loop coloring within each group.

Virtual matrix. As shown in Fig. 2, let the total number of nodes be ***n***, we imagine that all nodes are grouped to form the following virtual matrix (Taking 28 nodes as an example, according to Algorithm 1, get $g = 6$, $e = 5$, that is, 28 nodes are divided into 6 groups of 5 elements each):

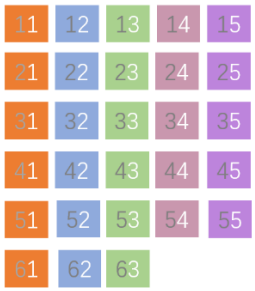

Fig. 2. Virtual matrix formed by all nodes

Note that this is just a virtual matrix we imagined, and it won't be stored, just to make it easier for us to understand the building process of the tree next.

LgTree. The core of LgTTBFT is the LgTree structure, a multi-reconstructed multi-way tree. Here, "*multiple-reconstructed*" means that each subtree stores all the nodes in the network in different structures. We store all nodes as ***e*** subtrees of different colors, each of which contains all nodes. The nodes in the ***color-root*** layer represent the colors of different subtrees, that is, different ***eIDs***; the node of the ***color-1*** layer represents the first node of each color tree; the node of the ***color-2*** layer represents the node of the current color remaining in each color tree (ie, the same node as the ***eID*** of the ***color-1*** layer node); Finally, the nodes of the ***leaf*** layer represent nodes of all other colors. First remove the ***color-1*** layer and ***color-2*** layer columns according to the position in the matrix, and then add the remaining nodes as children of the ***color-2*** layer node in rows, until they are added to ***e*** children, Fig. 3 shows the LgTree structure for 28 nodes. For space reasons, only two subtrees are drawn.

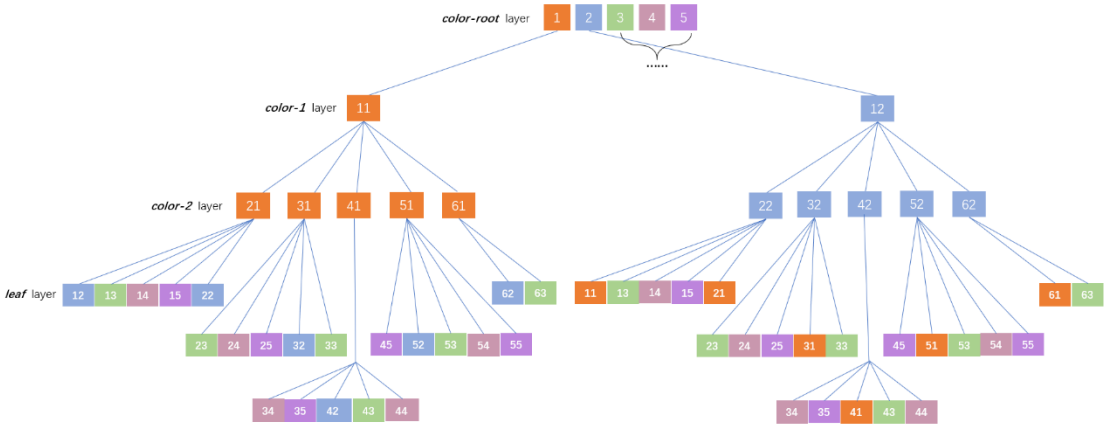

Fig. 3. LgTree structure (partial)

It is worth noting that if the total number of nodes is not the square of an integer, that is, the formed virtual matrix is not a square matrix (as shown in **Fig. 2**), then the nodes of the ***color-2*** layer of the incomplete column (the last two columns of the matrix) (nodes 24, 34, 44, 54 and nodes 25, 35, 45, 55) is unable to store all other color nodes (here nodes 61, 62, 63 can not be stored). At this point we store the remaining nodes in the last node of the ***color-2*** layer (such as nodes 54 and 55), which at most stores more ***e*** other nodes (when the matrix of $g = e$ is not full).

Storage. It can be seen from the construction of the multi-way tree that this is a tree with a small depth but a large width, and each subtree stores all the nodes, so if the entire tree structure is stored at each node, it will cause a large storage overhead. In this paper, we draw on the Kademlia algorithm and store only the node information associated with itself. Also, we only need to store the parent and child nodes of the nodes in the ***color-1*** layer and ***color-2*** layer in each subtree to store the entire tree structure.

4.2 Automated Construction

The construction process of the LgTree structure is given in Algorithm 2. The specific process is as follows:

1) Since the subtree we are constructing is ***e*** color trees, the outermost loop starts by traversing the element number ***e***. Here we are using the random number generation function in Golang. In order to make the random numbers generated each time different, we also provide different seeds for the generator (the current system time obtained). The random number generation here occurs in the TEE but is not publicly verifiable.

2) After selecting the ***eID***, start adding nodes one by one. For each node, we only store its parent and child nodes. If the node is a node of the ***color-1*** layer, then its parent is its ***gID***, child nodes is the same node as its own ***eID***.

3) For the nodes of the ***color-2*** layer, since the child nodes are added horizontally by the row, it is divided into whether or not to traverse from the first node of the virtual matrix. Only the first node of the ***color-2*** layer is added from the beginning, and the rest is marked with an *id* (*lastGroupID* in the algorithm) as the last position of the previous group, and then added from the next node.

Algorithm 2. Tree construction

Input: Number of groups, ***g*** ; Number of elements in each group, ***e***

Output: tree structure

```
1:  function Build ( g, e )
2:      lastGroupID ⟵ 0
3:      lastGroupGID ⟵ 0
4:      lastGroupGID ⟵ 0
5:      firstID ⟵ getFirstID(g,e)   ▷ the first id of the matrix
6:      firstGID ⟵ getFirstGID(firstID,g,e)   ▷ gID of the firstID
7:      firstEID ⟵ getFirstEID(firstID,g)   ▷ eID of the firstID
    ▷ Build the color tree in turn, starting from the eID
8:      for k = firstEID ; k <= e ; k++ do
9:          currentEID ⟵ k
10:         for j = firstGID ; j <= g ; j++ do
11:             currentGID ⟵ j
12:             if currentGID = = firstGID then   ▷ is node in color-1 layer
13:                 storeParent(currentGID,address)   ▷ parent node id is its'gID
14:                 for l = firstGID ; l <= g ; l++ do   ▷ Children's eID is the same as their own eID
15:                     storeChild(currentID_c1,address)   ▷ currentID_c1 = l
16:             else   ▷ is node in color-2 layer
17:                 groupLable:
                        ▷ start from firstID
18:                 if lastGroupID = = 0 then   ▷ determine whether to traverse from firstID
19:                     storeParent(currentID_c2_p ,address)   ▷ parent node gID is firstGid ,
                                                                 eID is the same as its own
                        ▷ store children nodes
20:                     for p = firstGID ; p <= g ; p++ do
21:                         for q = firstEID ; q <= e ; q++ do
22:                             if q = = currentEID then do nothing   ▷ is the id of the column
                                                                        crossed out in the matrix
23:                             else
                                    ▷ the number of child nodes stored in each color-2 layer node is e
24:                                 arrLenth ⟵ len(nodesInfoArr_c2)
25:                                 if arrLenth <= e+1 then   ▷ not yet full
26:                                     storeChild (currentID_c2,address )   ▷ currentID_c2's gID is p,
                                                                               eID is q
                                        ▷ record lastID
27:                                     lastGroupID ⟵ currentID_c2
28:                                     lastGroupGID ⟵ p ; lastGroupEID ⟵ q
29:                                     nodeID_parent ⟵ IDSplicing (j , i)   ▷ id splicing function
30:                                     storeLastGroupID (nodeID_parent, lastGroupID)
31:                                 else   ▷ should be stored in the next group
31:                                     break groupLable
                    ▷ not start from firstID , already have lastGroupID
31:                 else
32:                     storeParent(currentID_c2_p ,address)   ▷ currentID_c2_p's gID is
                                                                 firstGID, eID is currentEID
33:                     for s = lastGroupGID ; s <= g ; s++ do
34:                         for t = lastGroupEID ; t <= e ; t++ do
35:                             if t = = currentEID then do nothing   ▷ is the id of the column
                                                                        crossed out in the matrix
36:                             else
37:                                 arrLenth ⟵ len(nodesInfoArr_c2)
38:                                 if arrLenth <= e+1 then
39:                                     storeChild (currentID_c2,address )   ▷ currentID_c2's gID is s,
                                                                               eID is t
                                        ▷ record lastID
40:                                     lastGroupID ⟵ currentID_c2
41:                                     lastGroupGID ⟵ s ; lastGroupEID ⟵ t
42:                                     nodeID_parent ⟵ IDSplicing (j , i)
43:                                     storeLastGroupID (nodeID_parent, lastGroupID)
```

4.3 Simple and Efficient Routing

The routing algorithm for LgTTBFT is given in Algorithm 3. Each subtree stores all the nodes, so any node can be found in any subtree. The specific routing process is as follows:

1) In order to ensure load balancing, any subtree is randomly selected, that is, a random ***eID*** is generated within a certain range (the random value must not exceed the value of ***e***).

2) Then separate the ***gID*** of the node to be searched, and make a rough judgment based on the ***gID***. That is, the node with the same ***gID*** is found from the node of the ***color-2*** layer (the node is called *nodeC2*), and the child nodes of *nodeC2* are searched.

3) If the last node in *nodeC2*'s child node (called *lastChild*) happens to be the node to be searched, the search succeeds and the node is returned; if *lastChild* is greater than the id of the node to be searched, then look forward; otherwise, look in the next group.

Algorithm 3. Router

Input: The id of the node to search, ***nodeID***
Number of groups, ***g*** ; Number of elements in each group, ***e***
Output: node structure

```
1:  function Search ( nodeID, g, e )
2:      firstGID ⟵ getFirstGID (g)
3:      firstEID ⟵ getFirstEID (e)
    ▷ Randomly select a color tree, that is, randomly select an e
4:      randomEID ⟵ generateRand (seed)
5:      gID ⟵ getGID (nodeID)      ▷ gID of the node
    ▷ Query the node of the color2 layer, the GID of that node is the same as the GID
6:      if gID = = firstGID then   ▷ the color-2 layer node starts from gID=2
7:          gID = firstGID + 1
8:      else
9:          nodeID_c2 ⟵ IDSplicing (gID , randomEID)  ▷ ID of the color-2 layer node
10:         lastGroupID ⟵ getLastGroupID (nodeID_c2)  ▷ ID of the last node of the
                                                        children of nodeID_c2
11:         if lastGroupID = = nodeID then
12:             return node
13:         else if lastGroupID > nodeID then
14:             for j = 0 ; j <= e ; j++ do  ▷ to traverse the children node of
                                               the parent node (color2 layer node)
15:                 currentID ⟵ nodesInfoArr[j]
16:                 if currentID = = nodeID then
17:                     return node
18:         else    ▷ go to the next group to search
19:             gID_next ⟵ gID ++
20:             nodeID_c2 = IDSplicing (gID_next , randomEID)
21:             lastGroupID ⟵ getLastGroupID (nodeID_c2)
22:             if lastGroupID = = nodeID then
23:                 return node
24:             else
25:             for j = 0 ; j <= e ; j++ do
26:                 currentID ⟵ nodesInfoArr[j]
27:                 if currentID = = nodeID then
28:                     return node
```

4.4 Structural Advantages

Although the width of the tree structure is large and the nodes between the subtrees are redundant, each node does not store the entire tree structure. Only the nodes of the ***color-1*** layer and the ***color-2*** layer are stored in each subtree, and the parent node and the child nodes are stored. Thus for the entire tree, each node stores its associated nodes and there is no duplicate storage. In addition to storage efficiency, the LgTree architecture brings many performance and efficiency advantages:

- **Communication efficiency**. As can be seen from our routing algorithm, the node search efficiency is very high. At most, only *e+2* steps are needed to find the node, thus providing *O(n)* search efficiency.

- **Load balancing**. One problem with a single tree structure is that it does not guarantee a fair load distribution. It is possible that the upper node assumes all forwarding loads and the leaf nodes do not bear any forwarding load. According to the routing algorithm designed by our LgTree structure, we first randomly select a subtree when searching for nodes, and then search under the subtree, which gives all nodes equality. That is to say, the nodes in the subtree as ***color-1*** layer and ***color-2*** layer become ***leaf*** nodes in other subtrees. At the same time, the structure ensures the load balancing in each subtree while ensuring load balancing.
- **Reliability.** A tree structure consisting of a single tree is not sufficient to ensure reliability, and an upper node failure will result in the loss of data for all of its child nodes. The construction of LgTree's multiple subtrees improves the reliability of the system. Although it seems redundant, the storage overhead is small.

5. Trusted Execution Environment (TEE)

The introduction of the trusted execution environment brings about a reduction in the number of replicas and an increase in communication efficiency for the BFT consensus algorithm.

One possible failure in the Byzantine fault-tolerant algorithm is "*equivocates*", where a node sends a conflict proposal to another replica. As in PBFT, it is possible for a malicious *Leader* to send inconsistent messages to replicas. In order to prevent this possible failure, an additional communication step (PRE-PREPARE phase in PBFT) is required to verify that the messages sent by the *Leader* are consistent, thereby ensuring that all non-failure replicas execute the same proposal.

In addition to adding additional communication steps, another solution is to provide trusted services [30, 31, 16, 34], using a monotonically increasing trusted counter to securely assign a trusted counter value to each message and make the following guarantee: 1) Never bind the same counter value to a different message, 2) Never assign a value lower than the previous counter value (monotonically increasing), and 3) Never assign a discontinuous counter value (sequence). Therefore, in order to ensure that these three points can be achieved in any accidental or malicious situation, the counter must be implemented in a trusted component.

The number of replicas. In the traditional BFT fault-tolerant model, $3f+1$ replicas is required to tolerate f failure replicas only when the failure replica and the malicious replica are different replicas. In this paper, the introduction of the trusted counter avoids the occurrence of equivocates, that is, avoids the possibility of the replica to do evil, so the number of minimum replicas in BFT system is reduced from $3f+1$ to $2f+1$ by preventing equivocates.

Communication steps. The purpose of the PRE-PREPARE phase in the PBFT model is to verify the reliability of the messages sent by the *Leader*. This method of adding communication steps creates additional communication overhead. The use of a trusted counter to bind different messages to a monotonically increasing counter value ensures a consistent and irreversible modification of a broadcast message, thereby saving a communication step.

Communication complexity. In the three-phase protocol of the PBFT model, there are four message broadcasts, two of which are all-network full-node broadcasts, which causes two $O(n^2)$ communication complexity and consumes network bandwidth. The LgTTBFT algorithm reduces the communication overhead of the PREPARE phase to $O(n)$, which greatly improves communication efficiency.

In addition, TEE technology is also an effective way to solve the privacy problem of Blockchain. The fundamental reason for the challenge of blockchain privacy protection is usually that there is no trust assumption in the blockchain architecture, and the complexity of the solution is therefore very high. If in some scenarios we can accept trusted assumptions about technologies such as hardware, platform solutions will be more efficient, versatile, and secure. TEE technology (Trusted Computing Environment Technology) can help solve blockchain privacy issues through isolation and verifiability.

6. Evaluation

In this section, we implemented LgTTBFT, which simulates protocol execution under normal conditions and compares it to MinBFT. Our implementation is based on Golang and uses the Intel SGX to provide

hardware security support to implement the TEE part of LgTTBFT. Due to the limitation of experimental conditions, the test we carried out was single-machine simulation test. That is, using different ports to simulate different replicas, each running in a different process. Replicas were running on an Intel(R) Core(TM) i5-8265U CPU equipped with 16 GB RAM and Intel SGX.

For BBFT, which is also a hierarchical topology, because we don't have open source code, we don't have time to reproduce its experimental results, so we won't compare here.

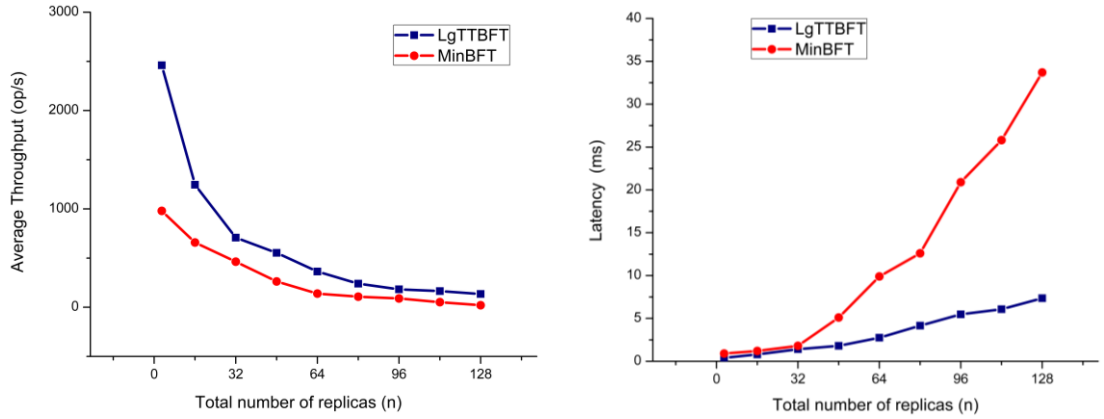

Fig. 4. Evaluation results for LgTTBFT and MinBFT

Performance for LgTTBFT and MinBFT. To illustrate the performance gains of structured networking, we chose to do a comparison with MinBFT because MinBFT also uses a trusted execution environment but does not use a structured network. Our results show that under normal circumstances, as the number of replicas increases, the throughput of LgTTBFT is always higher than that of MinBFT, and the latency is always smaller than MinBFT. As shown in Fig. 4, when $n \leq 48$, the throughput of LgTTBFT is more than twice that of MinBFT; when $n \geq 64$, the throughput of LgTTBFT is more than three times that of MinBFT. In terms of latency, the LgTTBFT rises steadily, with a latency of only 7ms in the case of 128 nodes. Therefore, the performance advantage of LgTTBFT is more obvious when the number of replicas is larger. The Ethereum, which is also a structured network blockchain, has a throughput of only 10-40 tps and performance is much lower than LgTTBFT.

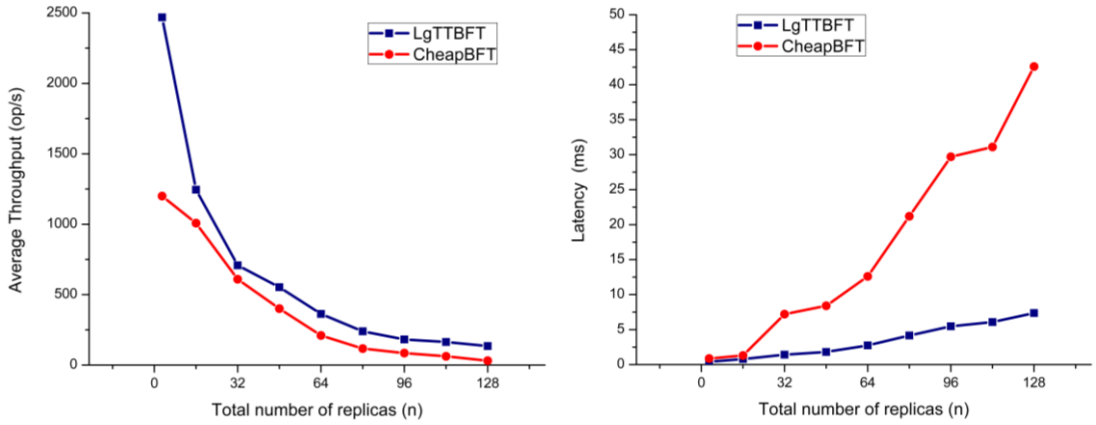

Fig. 5. Evaluation results for LgTTBFT and CheapBFT

Performance for LgTTBFT and CheapBFT. The two cores of CheapBFT are TEE and Optimistic BFT. This Speculation Byzantine protocol brings about a reduction in the number of executions and the number of protocol messages, thereby achieving resource savings. Compared with CheapBFT, our system does not adopt speculative strategy, but organizes nodes into a tree structure at the network layer to improve road efficiency. To illustrate the increased efficiency of the combination of structured networks and TEE, we conducted a comparative test with CheapBFT. Our results show that under normal circumstances, as the number of replicas increases, the throughput of LgTTBFT is always higher than that of CheapBFT, and the latency is always smaller than MinBFT, as shown in Fig. 5.

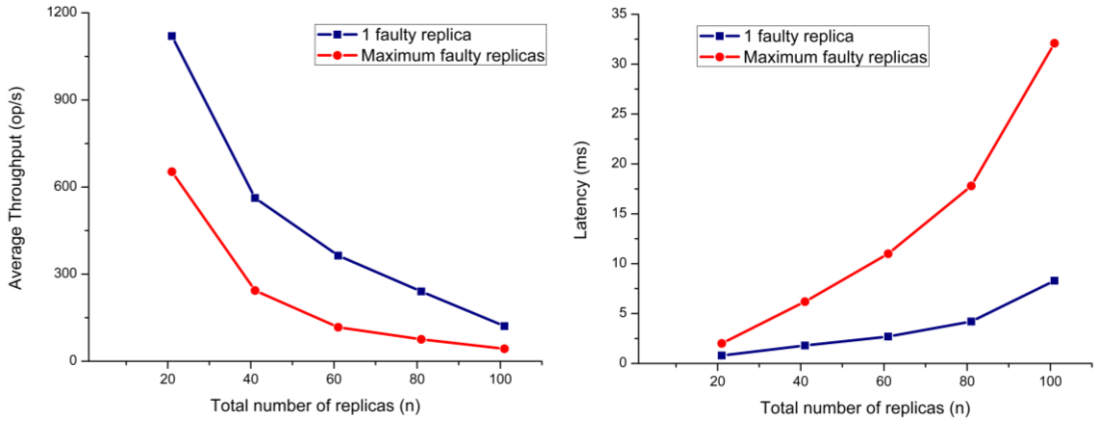

Fig. 6. Evaluation results for different faulty replicas

Impact of faulty replicas. We also evaluated the impact of the number of faulty replicas on the performance of the algorithm. Due to the introduction of TEE, a faulty replica cannot perform different operations on different correct replicas. The simplest failure scenario is a crash failure on the backup replicas, so the crash failure model we set here is offline for the faulty replicas. We set the number of faulty replicas to the maximum, that is, there are f faulty replicas in $2f+1$ replicas. We compared a single-faulty replica and f-faulty replicas under a certain number of replicas. The experimental results show that the number of faulty replicas has a large impact on the overall performance of the algorithm. As shown in Fig. 6, when $n \leq 41$, the throughput of the single-faulty replica algorithm is more than twice the algorithm throughput of f-faulty replicas; when $n \geq 61$, the throughput of the single-fault replica algorithm is more than three times the algorithm throughput of f-faulty replicas. Performance is significantly reduced when the number of faulty replicas reaches a maximum because the message must be collected for all remaining $f+1$ non-faulty replicas. Similarly, as the number of replicas increases, the performance advantages of LgTTBFT become more pronounced.

In this section we only tested a small, consensual portion, but in terms of our network architecture, Due to the provision of a structured P2P network (as described in Section 4), so scaling out to the entire system and larger node sizes should perform better.

7. Related Work

Recently, the Libra project [35] released by FaceBook has attracted widespread attention. The LibraBFT [36] consensus mechanism used in this project is a new BFT consensus mechanism based on HotStuff [37]. Compared with variants of classic PBFT algorithms such as Tendermint and Algorand, HotStuff combines with *chained structure* and uses *threshold signature* to reduce the complexity of the most expensive View-change step in the BFT algorithm to $O(n)$. The algorithm is more efficient and secure.

View-change in Byzantine fault tolerance has a lot of overhead and can achieve up to $O(n^4)$ complexity. Aardvark [38] is a Byzantine agreement in an attack situation designed to effectively tolerate Byzantine behavior and reduce the system's performance differences in the presence or absence of Byzantine errors. Based on the PBFT communication model, an adaptive mechanism is designed to ensure the security and liveness of the system: MAC-Signature Hybrid Authentication prevents clients from doing evil; more actively trigger view-change to avoid the Byzantine behavior of the *Leader*; P2P technology replaces broadcasting to improve communication efficiency; network separation (multiple network cards) to prevent traffic attacks. This algorithm provides fault tolerance rate of $3f+1$.

In terms of network liveness, HoneyBadgerBFT [25] is the first well-known asynchronous BFT protocol that can run in asynchronous networks where the message latency has no explicit upper limit. The protocol uses two methods to increase consensus efficiency: 1) Mitigating the bandwidth bottleneck of a single node by splitting transactions , and 2) Improve transaction throughput by selecting random trading blocks in batch transactions and matching threshold signatures. Compared with PBFT, the efficiency is significantly improved, and the fault tolerance rate is also $3f+1$.

The most prominent contribution of LibraBFT is that the complexity of the View-change step is reduced to $O(n)$. And what we did was to reduce the communication complexity from $O(n^2)$ to $O(n)$ in the Prepare phase. From this point of view, LibraBFT's contribution is even greater.

We didn't discuss View-change in this paper, and Aardvark is a protocol designed specifically for View-change scenarios, with the goal of reducing the system's differences in the presence or absence of Byzantium. This work is a good complement to the LgTTBFT algorithm and deserves our reference in future research.

LgTTBFT is a weakly synchronized BFT consensus protocol, and HoneyBadgerBFT is an asynchronous consensus protocol, which is the main difference between the two protocols. In terms of fault tolerance, the HoneyBadgerBFT system assumes that there is a reliable communication pipe connection between every two nodes, and the total number of nodes in the entire network must be greater than 1/3 of the faulty nodes. Therefore, HoneyBadgerBFT provides a fault tolerance of 3f+1, which is not as good as LgTTBFT.

8. Conclusion and Future Work

LgTTBFT proposes a new tree topology and routing algorithm, which achieves higher search efficiency and better reliability than MinBFT. The introduction of TEE further reduces the number of replicas of the protocol and improves communication efficiency. Experiments show that the effective combination of these two cores can provide better consensus efficiency and improve the current technical level.

There are also some imperfections in the agreement that are worth exploring in the future work. First, the view-change was not implemented. Only the normal running process was considered. The consensus efficiency when the Byzantine failure occurred was not optimized. Second, the tree construction algorithm is more complicated and worth further optimization.

References

1. Lamport L.The part-time parliament[J]. ACM Transactions on Computer Systems (TOCS), 1998, 16(2): 133 -169.
2. Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[C]//2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). 2014: 305-319.
3. Nakamoto S. Bitcoin: A peer-to-peer electronic cash system[J]. 2008.
4. King S, Nadal S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake[J]. self-published paper, August, 2012, 19.
5. Yang B, Garcia-Molina H. Improving search in peer-to-peer networks[C]// Proceedings 22nd International Conference on Distributed Computing Systems. IEEE, 2002: 5-14.
6. Gkantsidis C, Mihail M, Saberi A. Random walks in peer-to-peer networks[C]// IEEE INFOCOM 2004. IEEE, 2004, 1.
7. Stoica I, Morris R, Karger D, et al. Chord: A scalable peer-to-peer lookup service for internet applications[J]. ACM SIGCOMM Computer Communication Review, 2001, 31(4): 149-160.
8. Maymounkov P, Mazieres D. Kademlia: A peer-to-peer information system based on the xor metric[C]//International Workshop on Peer-to-Peer Systems. Springer, Berlin, Heidelberg, 2002: 53-65.
9. Intel, “Software Guard Extensions Programming Reference,” 2013. [Online]. Available: https: //software.intel.com/sites/default/files/329298-001.pdf
10. Lamport L, Shostak R, Pease M. The Byzantine generals problem[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1982, 4(3): 382-401.
11. Castro M, Liskov B. Practical Byzantine fault tolerance[C]//OSDI. 1999, 99(1999): 173-186.
12. Distler T, Kapitza R. Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency[C]//Proceedings of the sixth conference on Computer systems. ACM, 2011: 91-106.
13. Kotla R, Alvisi L, Dahlin M, et al. Zyzzyva: speculative byzantine fault tolerance[C]//ACM SIGOPS Operating Systems Review. ACM, 2007, 41(6): 45-58.
14. Kotla R, Dahlin M. High throughput Byzantine fault tolerance[C]//International Conference on Dependable Systems and Networks, 2004. IEEE, 2004: 575-584.
15. Veronese G S, Correia M, Bessani A N, et al. Efficient byzantine fault-tolerance[J]. IEEE Transactions on Computers, 2011, 62(1): 16-30.
16. Kapitza R, Behl J, Cachin C, et al. CheapBFT: resource-efficient byzantine fault tolerance[C]//Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012: 295-308.
17. Liu J, Li W, Karame G O, et al. Scalable byzantine consensus via hardware-assisted secret sharing[J]. IEEE Transactions on Computers, 2018, 68(1): 139-151.
18. NEO, “Consensus” 2017. [Online]. Available: https://docs.neo.org/docs/en-us/basic/technology/consensus.html
19. Mazieres D. The stellar consensus protocol: A federated model for internet-level consensus[J]. Stellar Development Foundation, 2015: 32.
20. Kwon J. Tendermint: Consensus without mining[J]. Draft v. 0.6, fall, 2014, 1: 11.
21. Kogias E K, Jovanovic P, Gailly N, et al. Enhancing bitcoin security and performance with strong consistency via collective signing[C]//25th {USENIX} Security Symposium ({USENIX} Security 16). 2016: 279-296.

22. ZILLIQA Team. The ZILLIQA Technical Whitepaper[J]. Retrieved September, 2017, 16: 2019. [Online]. Available: https://docs.zilliqa.com/whitepaper.pdf
23. Gilad Y, Hemo R, Micali S, et al. Algorand: Scaling byzantine agreements for cryptocurrencies[C]//Proceedings of the 26th Symposium on Operating Systems Principles. ACM, 2017: 51-68.
24. Ontology, “VBFT Consensus” 2018. [Online]. Available: https://dev-docs.ont.io/#/docs-en/DeveloperGuide/02-VBFT-introduction
25. Miller A, Xia Y, Croman K, et al. The honey badger of BFT protocols[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016: 31-42.
26. Ripeanu M. Peer-to-peer architecture case study: Gnutella network[C]// Proceedings first international conference on peer-to-peer computing. IEEE, 2001: 99-100.
27. Harren M, Hellerstein J M, Huebsch R, et al. Complex queries in DHT-based peer-to-peer networks[C]//International Workshop on Peer-to-Peer Systems. Springer, Berlin, Heidelberg, 2002: 242-250.
28. Stoica I, Morris R, Karger D, et al. Chord: A scalable peer-to-peer lookup service for internet applications[J]. ACM SIGCOMM Computer Communication Review, 2001, 31(4): 149-160.
29. Rowstron A, Druschel P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems[C]//IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer, Berlin, Heidelberg, 2001: 329-350.
30. Chun B G, Maniatis P, Shenker S, et al. Attested append-only memory: Making adversaries stick to their word[C]//ACM SIGOPS Operating Systems Review. ACM, 2007, 41(6): 189-204.
31. Levin D, Douceur J R, Lorch J R, et al. TrInc: Small Trusted Hardware for Large Distributed Systems[C]//NSDI. 2009, 9: 1-14.
32. Veronese G S, Correia M, Bessani A N, et al. EBAWA: Efficient Byzantine agreement for wide-area networks[C]//2010 IEEE 12th International Symposium on High Assurance Systems Engineering. IEEE, 2010: 10-19.
33. Wood T, Singh R, Venkataramani A, et al. ZZ and the art of practical BFT execution[C]//Proceedings of the sixth conference on Computer systems. ACM, 2011: 123-138.
34. Matetic S, Ahmed M, Kostiainen K, et al. {ROTE}: Rollback Protection for Trusted Execution[C]//26th {USENIX} Security Symposium ({USENIX} Security 17). 2017: 1289-1306.
35. Facebook, “Libra - Home” 2019. [Online]. Available: https://www.facebook.com/Libra/
36. Baudet M, Ching A, Chursin A, et al. State machine replication in the Libra Blockchain[J]. 2019.
37. Yin M, Malkhi D, Reiter M K, et al. Hotstuff: BFT consensus in the lens of blockchain[J]. arXiv preprint arXiv:1803.05069, 2018.
38. Clement A, Wong E L, Alvisi L, et al. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults[C]//NSDI. 2009, 9: 153-168.
39. Rihong W, Lifeng Z, Quanqing X, et al. K-buket based raf-like consensus algorithm for permissioned blockchain [C]// To appear in ICPADS. 2019.12