RB-Raft: A Byzantine Fault-Tolerant Raft Consensus Algorithm Postprint

Authors: Shuzhi Li, Zou Yijie, Deng Xiaohong, Luo Zhiqiong, Liu Huiwen

Date: 2022-05-11T10:48:43Z

Abstract

To address the issues that the Raft algorithm cannot resist attacks from Byzantine nodes and that logs are susceptible to tampering and forgery, we design an RB-Raft (Resist Byzantine-Raft) algorithm that resists Byzantine nodes. First, we employ a hash chain approach to perform iterative hashing on each log block, and verify logs through a dynamic verification mechanism, thereby providing a certain fault tolerance rate against malicious behavior of Leader nodes, which solves the problem of log forgery and verification. Second, we propose a "last will" mechanism based on threshold encryption, which legitimizes the process of Candidate nodes soliciting votes, preventing attacks where Byzantine nodes arbitrarily solicit votes to change the Leader node, and solves the problem of Byzantine nodes affecting system consistency. Experimental results demonstrate that the proposed RB-Raft algorithm possesses Byzantine resistance capability, with a log recognition rate reaching 100%. Meanwhile, compared to PBFT, the consensus latency of our algorithm is reduced by 53.3%, and the throughput is increased by 61.8%. The algorithm proposed in this paper is suitable for consensus in untrusted consortium blockchains.

Full Text

Preamble

RB-Raft: A Byzantine Fault-Tolerant Raft Consensus Algorithm

Li Shuzhi[1], Zou Yijie[1], Deng Xiaohong[2]†, Luo Zhiqiong[1], Liu Huiwen[2]
(1. College of Information Engineering, Jiangxi University of Science & Technology, Ganzhou, Jiangxi 341000, China;
2. School of Electronics & Information Engineering, Gannan University of Science & Technology, Ganzhou, Jiangxi 341000, China)

Abstract: To address the limitations of the Raft algorithm in resisting Byzantine node attacks and preventing log tampering, this paper proposes the RB-Raft

(Resist Byzantine-Raft) algorithm. First, we employ a hash chain approach to iteratively hash each log block, and through a dynamic verification mechanism, we enable fault tolerance for malicious behavior by Leader nodes, thereby solving the problems of log forgery and verification. Second, we introduce a "legacy" mechanism based on threshold encryption that legitimizes vote solicitation by Candidate nodes and prevents Byzantine nodes from arbitrarily pulling votes to replace the Leader, thus resolving the issue of Byzantine nodes affecting system consistency. Experimental results demonstrate that the proposed RB-Raft algorithm can effectively resist Byzantine nodes, achieving a 100% log recognition rate. Compared with PBFT, the algorithm reduces consensus latency by 53.3% while increasing throughput by 61.8%. The proposed algorithm is suitable for consensus in untrusted consortium blockchains.

Keywords: consensus mechanisms; Byzantine fault tolerance; hash chain; threshold encryption; "legacy" mechanism

0 Introduction

Since Nakamoto' s 2008 paper "Bitcoin: A Peer-to-Peer Electronic Cash System" [**?**], blockchain technology has continuously evolved as the underlying core technology of Bitcoin, giving rise to various types of blockchains. These include public blockchains such as Ethereum [**?**], consortium blockchains such as Hyperledger Fabric [**?**], and private blockchains such as Alibaba' s Ant Blockchain [**?**]. Blockchain is a decentralized, tamper-proof, and traceable distributed database system that integrates economics, P2P networks, consensus algorithms, asymmetric encryption, and other technologies—representing a highly integrated product of Internet technologies. As a distributed database system, the most critical challenge in blockchain is designing a consensus algorithm [**?**, **?**] to ensure data consistency among distributed nodes. Different blockchains employ different consensus algorithms: Bitcoin uses Proof of Work (PoW) [**?**], consortium blockchains generally adopt Practical Byzantine Fault Tolerance (PBFT) [**?**], while private blockchains typically use classical consistency algorithms such as Raft [**?**] and Paxos [**?**]. Among these, the Raft algorithm is widely applied in current engineering fields as a distributed protocol with strong consistency, high performance, and high reliability. Compared to Paxos, Raft is easier to understand and implement, but its inability to resist Byzantine nodes limits its application to private blockchains. Therefore, improving the Raft algorithm to make it applicable to consortium blockchains and even public blockchains has become a hot research topic.

The primary challenge in porting Raft to consortium blockchains lies in achieving Byzantine fault tolerance [**?**], which ensures that honest nodes can reach consensus despite interference from malicious nodes, guaranteeing normal system operation. PBFT reduces the computational complexity of Byzantine fault tolerance protocols from exponential $O(n(f+1))$ to polynomial $O(n^2)$, making

Byzantine protocols practical for distributed systems and becoming the mainstream consensus algorithm in current consortium blockchains. PBFT can tolerate at most (n-1)/3 faulty nodes, whereas the original Raft algorithm can tolerate at most (n-1)/2 crash-fault nodes. Moreover, Raft' s communication complexity of O(n) is far less than PBFT' s $O(n^2)$. Consequently, researchers have attempted to improve Raft to combine the advantages of both protocols.

In 2016, Christopher Copeland and Hongxia Zhong from Stanford proposed Tangaroa [**?**], a Byzantine fault-tolerant Raft that incorporates PBFT characteristics (including signed messages, malicious leader detection, and election verification) to achieve Byzantine fault tolerance while maintaining implementability and robustness. Reference [**?**] requires Follower nodes to sign votes to prevent forgery and mandates client signatures on logs to prevent Byzantine Leaders from forging logs, thereby enhancing Byzantine resistance to some extent. Reference [**?**] addresses the issues of vote splitting among multiple Candidate nodes and voting inefficiency caused by increasing Follower nodes in Raft' s leader election by using a double-layer Kademlia protocol to establish K-buckets for stable elections within Candidate sets. It also proposes a multi-Candidate parallel log replication scheme to balance Leader node load and address inefficiency in single-node log replication. Reference [**?**] adopts a two-level consensus mechanism that groups all nodes in Raft, with each group electing leaders to form a network committee. Raft consensus is used within groups, while PBFT is employed for consensus among committee members. However, this approach cannot effectively suppress Byzantine nodes within groups. Reference [**?**] transforms the vote-pulling process into a threshold signature process to prevent blank vote solicitation and introduces incremental hashing to ensure log immutability, though this increases log size and has limitations. Moreover, it cannot prevent Byzantine nodes from becoming Candidates and forcibly pulling votes to replace the Leader.

In summary, while Raft can be optimized to resist Byzantine nodes, several issues remain: (a) logs are easily tampered with and lack cryptographic protection; (b) vote casting lacks guarantees, as relying solely on term size for vote allocation is insecure, and votes are susceptible to forgery. To address these problems, we propose the RB-Raft (Resist Byzantine-Raft) algorithm with the following main innovations:

a) To prevent Byzantine Leader nodes from forging and tampering with logs, we implement a dynamic log verification mechanism based on hash chains, achieving tamper-proof logs and verifiable integrity.

b) To address vote forgery and the exploitation of votes by Byzantine nodes to replace the current Leader, we design a "legacy" mechanism based on threshold encryption. When the Leader node has not crashed, Follower nodes will not cast votes unless they obtain the Leader' s post-crash "legacy," which can only be acquired through threshold encryption. This prevents abnormal replacement of the Leader by Byzantine nodes.

1.1 Raft Workflow

The Raft algorithm defines three roles that nodes can assume, though a node cannot hold multiple roles simultaneously: Leader, Follower, and Candidate. Each role has distinct responsibilities: the Leader interacts with clients, synchronizes log information to Follower nodes, and maintains connections with Followers through HeartBeat messages; Followers respond to the Leader' s log synchronization requests and Candidate' s vote requests, storing the Leader' s log files locally—all nodes start as Followers; Candidates are responsible for election voting, transitioning to Leader status through voting when the Leader crashes. Raft begins by electing a Leader to manage log replication. The Leader accepts transaction requests (logs) from clients, replicates them to other cluster nodes, and notifies them to commit logs, ensuring all nodes remain synchronized. When the Leader crashes, other nodes initiate an election to select a new Leader. The Raft workflow is fundamentally determined by the Raft node state machine, as illustrated in Figure 1.

Raft employs a heartbeat mechanism to trigger re-election when the Leader crashes. Each Leader sends heartbeat signals to all cluster nodes to prove it is still alive. When a Leader crashes and cannot send heartbeats, Follower nodes that do not receive heartbeats within a specified timeout period switch to Candidate status, send election requests to other nodes, and receive votes if their logs are more up-to-date and their Term is larger; otherwise, nodes reject the vote request. If a Candidate receives more than n/2 votes (where n is the number of nodes), it becomes Leader and increments its Term. If a Leader heartbeat is received during the election process, the node reverts to Follower status.

1.2 Vote Forgery Problem

Similar to PBFT' s view concept, Raft uses a Term—a continuously increasing number representing the period during which a Leader holds authority. Each Term can produce only one Leader. Initially, all Followers have Term 1. When a Follower's logical clock expires without detecting a Leader heartbeat, it becomes a Candidate and increments its Term to 2. The node with the larger Term has higher priority and will not vote for a node with a smaller Term. If a Leader discovers a Follower with a larger Term, it automatically becomes a Follower. Essentially, each Term increment triggers a new election. During normal operation, all nodes share the same Term, which persists until a node fails. Requests with smaller Terms are rejected.

In Raft, the number of votes determines who becomes Leader because the node with more votes typically detected the Leader's crash earlier. Nodes uncondition-

ally vote for Candidates with larger Terms, and the algorithm selects Leaders based on vote count to quickly restore normal cluster operation. However, since a node's role is determined by its state machine, Byzantine nodes can arbitrarily change their identity. When the current Leader has not crashed, a Byzantine node can become a Candidate, increase its Term, solicit votes from other nodes, and replace the legitimate Leader upon obtaining sufficient votes.

1.3 Log Forgery Problem

Log replication is central to Raft, ensuring data consistency across the cluster. Only the Leader accepts client requests and forwards log entries to Followers. The Leader never deletes logs, and Followers only accept logs from the Leader. The log structure is shown in Figure 2, where x and y represent commands. Logs consist of an index and entries, with each entry containing a Term and command. The committed range includes logs accepted and stored by more than half of the nodes.

In Raft, each entry is uniquely identified by its index and Term, with all preceding entries being consistent. When a Leader crashes and a new Leader is elected, all nodes synchronize to the new Leader' s local log structure, keeping only the portion with identical indexes and Terms—excess entries are deleted and missing ones are synchronized to maintain consistency.

However, Raft Followers unconditionally accept and replicate the Leader' s log structure, even if their own logs are more recent. Since log messages are packaged and disseminated by the Leader, a Byzantine Leader could tamper with client-submitted logs. Therefore, ensuring log content cannot be forged or tampered with is crucial for Byzantine resistance.

2.1 Hash Chain-Based Dynamic Log Verification Mechanism

The concept of "hash chain" was first proposed by Lamport in 1981 in "Password authentication with insecure communication" [**?**] to prevent password theft and tampering during transmission. Hash chains use hash functions for multiple iterative encryptions, offering strong interference resistance while allowing servers to verify entire ciphertext sequences by storing only the final hash. Since log structures require immutable order and content, we use hash chains to link all log blocks, enabling verification of all previous blocks by checking only the final hash value.

Log Hash Chain Generation: When clients batch logs, they perform hash chain operations on log blocks. The generation process is illustrated in Figure

3, where Log represents the log block body and Proof represents the verification hash. The algorithm is described as follows:

Algorithm 1: Construct Log Hash Chain
Input: Log block sequence Log[]
Output: Log hash chain table ProofTable
1. Int i, j = 1 // i is the log loop index, j is the proof loop index
2. while (i != MaxLength(Log)) // Continue until hash chain covers all logs
3. string file = getLogBlockToFile(Log[i]) // Retrieve log
4. int proof // Define new proof for each iteration
5. if i == 1 // Check if first log block
6. proof = MD5toFile(file);
7. ProofTable[j] = proof;
8. j++, i++;
9. break; // Enter next loop
10. end if
11. proof = MD5toFile(file);
12. ProofTable[i] = HybridMD5(proof, ProofTable[i-1]);
13. j++, i++;
14. break;
15. end while

After generation, Follower nodes must verify received log blocks. In our approach, Follower nodes send verification requests to clients. When verification is needed, a Follower computes the hash chain of its stored logs to obtain the Proof of the last block, which represents all logs in the node. It then sends the last log block index as a request parameter to the client, which returns the corresponding Proof for comparison. If comparison fails, indicating inconsistency, the node rolls back to the previous Proof and repeats the comparison until a match is found. Incorrect Proof logs are deleted, and the node requests synchronization of the corrupted logs from the client.

Dynamic Log Verification Mechanism: Log comparison failures may result from: (1) the current Leader being Byzantine and modifying logs; (2) a previous Byzantine Leader that went undetected; or (3) transmission anomalies such as network fluctuations or I/O errors causing log loss or corruption. Since the cause cannot be determined, the Leader node cannot be immediately rejected when errors occur. Therefore, we design the following dynamic verification mechanism for fault tolerance:

Each Follower sends detection requests to the client every T logs, returning the Proof corresponding to its current local log index and performing iterative hashing for verification. Leveraging the logarithmic function's rapid convergence near 1, we design the piecewise function shown in Equation (1): T_n represents the T value in round n, and x acts as a credit value initialized to 1 with a step size of 0.1.

When verification fails, detection frequency should increase (i.e., T should de-

crease). For example, with x initially 1, when an error is detected, x first decreases by 0.1. When negative and rounded up, 2^x becomes a decreasing function. When x < 1, T_n becomes smaller than T_{n-1}, shortening the detection span and increasing frequency. Conversely, when verification succeeds, x increases by 0.1. When x > 1, 2^x becomes an increasing function, causing T_n to exceed T_{n-1}, reducing detection frequency to avoid overloading the client. When T reaches 1, meaning the Follower verifies after each log synchronization, the Leader is considered untrustworthy. When half of the Followers reach this state, the client replaces the Leader and initiates a new election.

The decision to replace the Leader depends on three factors: T value, the number of erroneous logs sent by the Leader, and cluster synchronization speed. While the latter two cannot be controlled, the initial T value can be set to determine the mechanism' s strictness. The T value decreases at a rate of 2^x. A larger T permits higher error rates, while a smaller T enforces stricter verification.

2.2 Threshold Encryption-Based "Legacy" Mechanism

A "legacy" traditionally refers to a public document specifying how to distribute a deceased person' s assets. In Raft, when the Leader crashes, Follower nodes decide whether to cast votes based solely on Term size, which is determined by whether they receive heartbeats from the Leader. This is easily exploited by Byzantine nodes that can modify their Term to obtain votes and replace the legitimate Leader.

To solve this problem, we design a "legacy" mechanism where a Leader generates a legacy (will) upon election, and only nodes obtaining this legacy can solicit votes from Followers (asset distribution). The legacy content can only be opened after the Leader crashes. To ensure security, we encrypt the legacy using threshold encryption, proposed by Desmedt and Frankel [**?**]. Threshold cryptography distributes key information among multiple users; decryption requires more than the threshold number of key shares—fewer than the threshold cannot decrypt. Only when most Follower nodes provide their key shares can the legacy be decrypted and vote solicitation proceed. Whether to provide key shares depends on receiving the Leader' s heartbeat (verifying Leader liveness), forming a complete verification loop that prevents Byzantine nodes from replacing the Leader by modifying Terms.

Legacy Generation Phase: The Leader first uses KeyGen with security parameter , user count n, and threshold t (typically set to half the cluster size) to generate public key pk and private key shares sk = (sk_id1, ···, sk_idn). It then encrypts the legacy message Lmsg using Enc(pk, Lmsg) to obtain ciphertext L.

Key and Legacy Distribution Phase: The Leader broadcasts the legacy (L, the encrypted ciphertext), each Follower' s corresponding private key share sk_idj, and the hash value HashLegacy of Lmsg.

Legacy Decryption Phase: When a Follower' s heartbeat timer expires without receiving a heartbeat, it obtains decryption shares L_id from other nodes to perform threshold decryption on L using Dec(sk_id, L). The node must collect t shares and combine them using Combine(L_id1, ···, L_idj) to recover Lmsg. When receiving threshold requests, nodes check if the time since the last heartbeat exceeds M; only then will they provide their private key share. If the Leader has not crashed, some nodes will have received heartbeats within M time, preventing key share distribution. M is set to 2/3 of the Follower' s heartbeat timeout, as experiments show most nodes receive heartbeats within the first 2/3 of the timeout period.

Vote Solicitation Phase: After obtaining the legacy content, the node becomes a Candidate and sends RequestVote messages to other Followers. Vote recipients must verify the Candidate's Lmsg by hashing it and comparing it with the previously received HashLegacy from the Leader. Only if the verification passes can normal voting proceed; otherwise, no vote is cast.

The legacy generation and decryption process is illustrated in Figure 4. The subsequent normal vote solicitation phase was introduced in Section 2.1.

Algorithm 2: Legacy Generation
Input: Lmsg, n, t,
Output: sk, pk, L
1. if node.status = Leader and flag = true then // Node becomes Leader
2. pk, sk = KeyGen(, n, t) // Generate key pair
3. L = Enc(pk, Lmsg) // Encrypt legacy
4. HashLegacy = h(Lmsg)
5. Broadcast(pk, sk, L, HashLegacy) // Broadcast legacy
6. end if

Algorithm 3: Threshold Share
Input: L, sk_id
Output: L_id
1. if node.status = Follower and getLMessage then // Follower requested for key share
2. if T > timeout then // If time exceeds timeout
3. return L_id = Dec(sk_id, L) // Provide key share
4. else
5. return err

Algorithm 4: Legacy Decryption
Input: L_id1, ···, L_idj, L
Output: Lmsg
1. if node.status = Follower and heartbeat = false then // Follower received no heartbeat
2. if count(L_id1, ···, L_idj) t then
3. Lmsg = Combine(L_id1, ···, L_idj) // Decrypt legacy
4. node.status = Candidate

5. return Lmsg
6. else
7. return err

Algorithm 5: Legacy Verification
Input: Lmsg, HashLegacy
Output: vote
1. if node.status = Follower and getLmsg then // Received vote request
2. if HashLegacy == h(Lmsg) then
3. Proceed to normal voting judgment
4. if judgment passes then
5. return vote
6. return err

---

3 Security Analysis

As a Byzantine-resistant Raft algorithm, RB-Raft must satisfy the following security requirements: (1) Logs cannot be modified by the Leader—ensuring consistency across all nodes is the most critical property of a consensus algorithm; (2) The Leader must not be abnormally replaced through malicious vote solicitation by Candidates.

3.1 Log Integrity

Ensuring log integrity requires: (1) Immutable log order; (2) Identical log blocks. We define log block B_i where i > 0 and i , and hash sequence H_j where j > 0 and j , with i, j .

Definition 2: h(Text) is a hash function (MD5 in this paper) that returns a fixed-length hash value.

If a malicious Byzantine node modifies log block B_k to B' _k, the resulting hash sequence H' _j is:

$$H'_j = h(h(\cdots h(h(B'_k), B_{k-1}), \cdots), B_1)$$

Therefore, there must exist $H'_j \neq H_j$, indicating that tampering with any log block in the sequence changes H_j. Upon detection, the corrupted log can be synchronized from the client, ensuring log integrity and authenticity.

3.2 Node Election Security

In Raft, the Leader periodically sends AppendEntries heartbeats to Followers to declare its liveness. When the Leader crashes, Followers that fail to receive AppendEntries within their heartbeat timeout increment their Term and become Candidates to elect a new Leader. In Byzantine environments, Candidate

vote solicitation lacks restrictions—malicious Candidates can replace the current Leader even if it hasn' t crashed (since their larger Term triggers vote casting). To prevent this, our threshold signature-based "legacy mechanism" ensures that Byzantine nodes must exceed threshold t (typically n/2) to obtain the legacy. This guarantees that vote solicitation is justified by evidence of Leader failure, ensuring election security.

3.3 Byzantine Resistance Capability Analysis

Raft is a private blockchain consensus algorithm that tolerates only crash faults, not Byzantine nodes. RB-Raft, improved for consortium blockchains, permits Byzantine nodes. We compare it with the mainstream PBFT algorithm.

Byzantine resistance depends on the maximum tolerable Byzantine nodes f in a cluster of n nodes. In RB-Raft, following the majority principle, normal nodes only need to outnumber Byzantine nodes by one, giving n = 2f + 1 and maximum tolerance f = (n - 1)/2. In PBFT, assuming faulty and Byzantine nodes are distinct, there are f faulty nodes and f Byzantine nodes. After excluding faulty nodes, normal nodes must outnumber Byzantine nodes by one, giving n = 3f + 1 and maximum tolerance f = (n - 1)/3. Therefore, RB-Raft tolerates (n - 1)/6 more Byzantine nodes than PBFT, demonstrating stronger Byzantine resistance.

---

4 Simulation Experiments

We implemented RB-Raft in Go, simulated consensus processes with multiple nodes on a single machine, and recorded throughput, consensus latency, and Byzantine behavior data for comparative experiments. Results demonstrate RB-Raft' s Byzantine resistance, high throughput, and low latency advantages.

4.1 Byzantine Resistance Performance Testing

This section introduces controllable Byzantine nodes to test cluster resistance, conducting experiments with different identities and behaviors.

1. Log Anti-Forgery Testing: We first test log anti-forgery capabilities. Due to election randomness, we set the election timeout of controllable nodes much smaller than normal nodes but longer than normal communication time, ensuring they become Leaders first during cluster initialization. We then send 500 identical logs from the client to the Leader, which uses a pseudo-random generator to modify 10 logs with indices in [1, 500]. We monitor whether Followers can correctly detect and roll back tampered logs. Experiments were conducted on both Raft and RB-Raft clusters, with results shown in Table 1.

Table 1: Log Security Test Results | Tampered Log Indices | Detected Tampered Log Indices | |————-|————-| | Raft:

383,11,484,103,133,211,371,368,75,320 | None detected | | RB-Raft: 327,33,451,440,30,179,98,117,254,413 | 327,33,451,440,30,179,98,117,254,413 (100% detection) |

Raft completely fails to detect tampered logs, synchronizing all logs from the Leader—clearly unacceptable in Byzantine environments. RB-Raft accurately detects all randomly tampered logs and performs rollbacks, achieving 100% accuracy and demonstrating excellent resistance to log forgery.

We also test the dynamic verification mechanism' s ability to quickly replace malicious Leaders while reducing detection frequency for normal Leaders to alleviate verification pressure. From a Follower' s perspective, we test T-value curves for both Byzantine Leaders (modifying logs) and normal Leaders, with initial T = 8. Results are shown in Figure 5.

Figure 5: T-Value Change Curve Comparison

Normal Leaders increased T to 16 after round 11, reducing detection frequency and alleviating node pressure. Byzantine Leaders caused T to rapidly decrease to 1 by round 2 after log modification, reaching an untrusted state and enabling replacement. This behavior stems from the piecewise function in Equation (1), where x acts as a credit value affecting T through its increments and decrements.

2. Legacy Mechanism Testing: We again introduce controllable malicious nodes and have them solicit votes while the Leader remains alive in both Raft and RB-Raft clusters. After 20ms, we count received Follower votes, with results in Tables 2 and 3.

Table 2: Legacy Mechanism Test with 100 Nodes | Algorithm | Votes Received | Threshold Shares | Legacy Obtained? | |——-|————-|————|————| | Raft | 51 | N/A | Yes | | RB-Raft | 0 | 15 (insufficient) | No |

Table 3: Legacy Mechanism Test with 500 Nodes | Algorithm | Votes Received | Threshold Shares | Legacy Obtained? | |——-|————-|————|————| | Raft | 251 | N/A | Yes | | RB-Raft | 0 | 72 (insufficient) | No |

Raft cannot contain Byzantine behavior—nodes obtain majority votes even when the Leader hasn' t crashed, enabling replacement. RB-Raft receives zero votes because threshold shares (15% and 14.4%) are below the threshold, preventing legacy acquisition and thus blocking malicious vote solicitation.

4.2 Consensus Latency

Consensus latency, measured as the time from client command issuance to receiving the cluster' s synchronization completion command, is a critical metric. Figure 6 compares PBFT, Raft, and RB-Raft latencies.

Figure 6: Consensus Latency Comparison

PBFT exhibits higher latency due to its multi-phase communication complexity. RB-Raft' s latency is slightly higher than Raft because of added key distribution

and legacy mechanisms, but remains within acceptable ranges. Thus, RB-Raft sacrifices some efficiency for Byzantine resistance.

4.3 Throughput Testing

Throughput, measured as TPS (Transactions Per Second), is a key blockchain performance indicator primarily affected by consensus efficiency. Using the EOSBenchTool and controlling variables with only the algorithm changing, we compare Raft, PBFT, and RB-Raft throughputs, selecting 20 random test groups shown in Figure 7.

Figure 7: Throughput Test Comparison

RB-Raft and Raft both achieve high throughput. RB-Raft' s throughput is 39.1% lower than Raft but 61.8% higher than PBFT. Byzantine fault tolerance requires mutual identity verification and increased communication, reducing throughput relative to Raft but maintaining superiority over PBFT.

4.4 Algorithm Performance Comparison

Table 4: Algorithm Performance Comparison | Algorithm | Max Byzantine Nodes (of n) | Avg. Latency (ms) | Avg. Throughput (TPS) | Communication Overhead | |———-|————————|————-|————————|————————| | PBFT | (n-1)/3 | $O(n^2)$ | - | - | | Raft | 0 | 2(n-1) | - | - | | RB-Raft | (n-1)/2 | 3(n-1) | - | - |

RB-Raft tolerates the most Byzantine nodes ((n-1)/2), showing strongest resistance. Its latency is lower than PBFT but slightly higher than Raft, and throughput exceeds PBFT while remaining below Raft. Communication overhead is $O(n^2)$ for PBFT, but only O(n) for both Raft and RB-Raft.

Table 5: Comparison with Byzantine Fault-Tolerant Raft Variants | Algorithm | Max Byzantine Nodes | Leader Election Time | Log Encryption | Vote Verification | |———-|————————|————————-|————-|————-| | RB-Raft | (n-1)/2 | Same as Raft | Yes | Yes | | Reference [13] | (n-1)/2 | Same as Raft | Only vote signing | No | | Reference [14] | 4f f +2f +f (grouped) | Faster (Kademlia) | No | No | | Reference [15] | Depends on PBFT | Same as PBFT | No | No | | Reference [16] | (n-1)/2 | Same as Raft | Incremental hash | Threshold signature |

RB-Raft matches reference [13] in Byzantine tolerance but provides more comprehensive improvements in log encryption and vote verification. Reference [14] achieves faster election through Kademlia but lacks Byzantine resistance. Reference [15]' s grouping approach inherits PBFT' s fault tolerance without significantly improving Raft. RB-Raft offers balanced enhancements in security and performance.

---

5 Conclusion

This paper proposes RB-Raft, a Byzantine-resistant Raft algorithm that solves log tampering and Byzantine fault tolerance issues in uncertain environments. We employ hash chains for log processing with dynamic verification to prevent forgery while reducing verification overhead. Our threshold encryption-based "legacy" mechanism uses heartbeat verification as proof to prevent Byzantine nodes from replacing the Leader through vote solicitation, ensuring system consistency. Experimental data show RB-Raft achieves 100% log recognition, 53.3% lower consensus latency than PBFT, and 61.8% higher throughput. RB-Raft is suitable for consensus in untrusted consortium blockchain environments requiring high efficiency and security, such as vehicular and IoT networks. Future work will focus on applying this algorithm to vehicular and IoT scenarios.

References

[1] Nakamoto S. Bitcoin: A peer to peer electronic cash system [EB/OL]. 2008. https://bitcoin.org/bitcoin.pdf.
[2] Ferretti S, D' Angelo G. On the ethereum Blockchain structure: A complex networks theory perspective [J]. Concurrency and Computation: Practice and Experience, 2020, 32(12): e5493.
[3] Wang G, Zhang S, Yu T, et al. A systematic overview of Blockchain research [J]. Journal of Systems Science and Information, 2021, 9(3): 257-275.
[4] Li W, He M, Haiquan S. An overview of Blockchain technology: applications, challenges and future trends [C]// 2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC). IEEE, 2021: 31-39.
[5] Arnold R, Longley D. continuity: a deterministic byzantine fault tolerant asynchronous consensus algorithm [J]. Computer Networks, 2021, 199(11): 108431-108443.
[6] 邓小鸿, 王智强, 李娟, 等. 主流区块链共识算法对比研究 [J]. 计算机应用研究, 2022, 39(1): 1-8. (Deng Xiaohong, Wang Zhiqiang, Li Juan, et al. Comparative research on mainstream Blockchain consensus algorithms [J]. Application Research of Computers, 2022, 39(1): 1-8.)
[7] Meneghetti A, Sala M, Taufer D. A survey on pow-based consensus [J]. Annals of Emerging Technologies in Computing (AETiC), 2020, 4(1): 8-18.
[8] Li Y, Wang Z, Fan J, et al. An extensible consensus algorithm based on PBFT [C]// 2019 International conference on cyber-enabled distributed computing and knowledge discovery (CyberC). IEEE, 2019: 17-23.
[9] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm [C]// 2014 USENIX Annual Technical Conference (USENIX ATC 14). 2014: 305-319.
[10] Lamport L. Fast paxos [J]. Distributed computing, 2006, 19(2): 79-103.
[11] Lamport L, Shostak R, Pease M. The byzantine generals problem [M]// Concurrency: the Works of Leslie Lamport. 2019: 203-226.

[12] Copeland C, Zhong H. Tangaroa: a byzantine fault tolerant raft [J]. Stanford University. 2016.
[13] Tian S, Liu Y, Zhang Y, et al. A Byzantine Fault-Tolerant Raft algorithm combined with schnorr signature [C]// 2021 15th International Conference on Ubiquitous Information Management and Communication (IMCOM). IEEE, 2021: 1-5.
[14] 王日宏, 周航, 徐泉清, 等. 用于联盟链的非拜占庭容错共识算法 [J]. 计算机科学, 2021, 48(09): 317-323. (Wang Rihong, Zhou Hang, Xu Quanqing, et al. Non-byzantine fault tolerance consensus algorithm for consortium Blockchain [J]. Computer Science, 2021, 48(09): 317-323.)
[15] 黄冬艳, 李浪, 陈斌, 等. RBFT: 基于 Raft 集群的拜占庭容错共识机制 [J]. 通信学报, 2021, 42(03): 209-219. (Huang Dongyan, Li Lang, Chen Bin, et al. RBFT: a new Byzantine fault-tolerant consensus mechanism based on Raft cluster [J]. Journal on Communications, 2021, 42(03): 209-219.)
[16] 王日宏, 张立锋, 周航, 等. 一种结合 BLS 签名的可拜占庭容错 Raft 算法 [J]. 应用科学学报, 2020, 38(01): 93-104. (Wang Rihong, Zhang Lifeng, Zhou Hang, et al. A byzantine fault tolerance raft algorithm combines with BLS signature [J]. JOURNAL OF APPLIED SCIENCES–Electronics and Information Engineering, 2020, 38(01): 93-104.)
[17] Lamport L. Password authentication with insecure communication [J]. Communications of the ACM, 1981, 24(11): 770-772.
[18] DESMEDT Y, FRANKEL Y. Threshold cryptosystems [C]. In: Advances in Cryptology–CRYPTO' 89. Springer Berlin Heidelberg, 1989: 307-315. [DOI: 10.1007/0-387-34805-0_28]

Note: Figure translations are in progress. See original paper for figures.