

Article

Block-as-a-Vote: A Framework Scaling Voting-Based Blockchain Consensus Protocols from Single-Model to Multi-Model

Hang Xiong  and Jing Li *

School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China; flybear@mail.ustc.edu.cn

* Correspondence: lj@ustc.edu.cn

Abstract

Blockchain consensus protocols lie at the core of blockchain systems and define how nodes process and finalize block confirmation, provide security guarantees for the system, and meanwhile influence the system performance. Improving the performance of blockchain consensus protocols while ensuring system security is a challenge. Many blockchain consensus protocols, while providing rigorous security guarantees, exhibit low performance. The primary reason is that they process blocks in a serial manner (named single-model), resulting in inefficient utilization of node resources. Nevertheless, one category of blockchain consensus protocols confirms blocks through voting, and they exhibit concurrency in the processing of blocks and votes. In this paper, we propose block-as-a-vote (BaaV), a framework that extends the voting-based blockchain consensus protocols from single-model to multi-model. From the perspective of data structure, the BaaV framework extends the vote into a block that carries data, thus enabling parallel block generation. It allows a block to integrate information from other blocks and votes, thereby achieving batch confirmation of blocks. More importantly, the BaaV framework still restricts votes to voting messages in terms of functionality, and when nodes vote and determine whether a block can be confirmed, they follow the original blockchain consensus protocols, thereby preserving the original security. The BaaV framework can significantly improve the resource utilization of nodes to achieve better performance. In these experiments, the BaaV framework improves the upper bound of node *TPS* from approximately 2 MB/s to about 6 MB/s, with only a marginal increase in *Latency*.

Keywords: blockchain consensus protocol; distributed consistency; resource optimization; performance optimization



Academic Editor: Gianluca Lax

Received: 13 September 2026

Revised: 8 October 2026

Accepted: 8 October 2026

Published: 10 October 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Ensuring data trustworthiness has long been a challenging research topic, and the emergence of blockchain technology [1] offers a new solution to this problem. A blockchain system is a distributed system composed of multiple mutually untrusted nodes. It organizes data in the form of blocks, each of which saves the hash value of other blocks, thereby forming a chain structure (known as the blockchain). In the blockchain system, each node maintains a copy of the chain structure and only trusts its own copy. Under this design, any modification to a block within a node's chain structure can be detected by the hash verification of the entire chain structure and be detected by other nodes. As a result, applications built upon the blockchain system exhibit decentralization and data reliability.

In the blockchain system, when a user calls the upper-layer application, the corresponding data (named the transaction) is first packaged into a block by a node and broadcast across the network. Then, all nodes confirm the block and update their data state according to the transactions in the block. Throughout this process, the blockchain system must address the consensus problem, also referred to as the distributed consistency problem [2]: in a network of N nodes with a fraction f of malicious nodes, all honest nodes must confirm the same block to maintain the same data state. The blockchain system contains a core module, the blockchain consensus protocol (BCP) [3–8], which addresses this consensus problem. The BCPs define the behavior of nodes throughout the entire process, including how nodes generate new blocks and exchange messages to achieve block confirmation. It can be observed that the execution of a BCP fully overlaps with block processing in a blockchain system. Therefore, the performance of BCPs directly influences the overall system performance. Meanwhile, the BCPs serve as the security core of the entire blockchain system. Hence, designing secure and efficient BCPs is critical to the usability of blockchain systems. Currently, while some BCPs offer rigorous security guarantees, their performance often falls short of efficiency.

A BCP is typically implemented as a periodically repeated process to continuously serve upper-layer applications. In each period, all honest nodes follow the setting in the BCP to generate blocks, exchange information for block confirmation (named the confirmation message), and confirm blocks based on the received confirmation message. This paper uses voting to refer to the process of generating confirmation messages. Correspondingly, the BCP can be logically separated into three mechanisms that define the block generation, voting, and block confirmation behaviors of nodes.

A mainstream direction for improving the BCPs' performance is to design a more efficient confirmation mechanism to reduce the message complexity for confirming blocks [3–5,7–11]. This direction is closely aligned with optimizing the message complexity of the traditional Byzantine-fault-tolerant (BFT) protocols, which is an inherently challenging problem. In the blockchain context, the scale of nodes is typically large, which will slow down the message propagation and further delay block confirmation. On the other hand, improving the performance of the BCPs also requires a sound overall design. Simple parameter adjustments, such as block size and block generation interval, may affect the overall network load without enhancing performance. This paper considers the performance of BCPs from the perspective of resource utilization. A BCP utilizing more node resources can support a greater load from upper-layer applications, leading to better performance. From this perspective, existing BCPs can be categorized into two models: single-model [3–12] and multi-model.

The process of the single-model BCPs can be idealized as generating and confirming one block in each period. Clearly, it results in significant idle resources of nodes when they have not received the block. Note that, in practice, to ensure that at least one block is generated in each period, some BCPs select more than one node as the block-generator [3,4,6], potentially leading to the generation of several blocks. However, only one of these blocks will be confirmed; thus, the resource utilization is not improved. The advantage of the single-model BCPs lies in its simple design, and it is easy to provide rigorous security guarantees, which is critical in blockchain systems. The multi-model BCPs try to enable nodes to generate and confirm a batch of blocks in each period. The multi-model BCPs have a higher utilization of node resources, but the cost is that the design is much more complex, especially in the security guarantees.

By reviewing related work, we find that the multi-model BCPs achieve performance advantages over their single-model counterparts because they enable nodes to generate blocks concurrently and confirm blocks in batches, thereby reducing the number of idle nodes. However, many single-model BCPs complete block confirmation via voting (named

voting-based *BCPs*). We observe that although the voting-based *BCPs* process blocks following the single-model, they exhibit concurrency in generating both blocks and votes, providing opportunities for further optimization. Moreover, the essence of the voting-based *BCPs* is to require nodes to select the same element from a state set through voting. Honest nodes choose the element based on the voting results. Under the same voting result, the form of the element does not affect the final selection. This means that the voting-based *BCPs* can inherently vote on and confirm multiple blocks simultaneously. What is needed is a method to integrate multiple blocks into a single unit.

Based on the above two characteristics, we propose the block-as-a-vote (BaaV) framework, which can scale the voting-based *BCPs* from single-model to multi-model. In the BaaV framework, the vote is extended to have the same data structure as a block and contains data, enabling the generation of a batch of blocks in each period. Each block contains an additional field to record the information of multiple blocks and votes. The additional field logically integrates multiple data-carrying items into a single unit, but it does not impact the security-related operations of the *BCPs*. When nodes vote or determine whether a block can be confirmed, they ignore the additional field and strictly adhere to the original *BCPs*, thereby theoretically inheriting the original security. When confirming a block, the additional field enables nodes to perform batch confirmation of blocks. Experimentally, the BaaV framework can achieve a nearly linear increase in *TPS* when the data load does not exceed node resources, with only a marginal increase in *Latency*. BaaV mainly augments the block data of the voting-based *BCPs* and introduces no additional complex operations as a whole. Therefore, compared with existing multi-model protocols, the protocols extended by BaaV remain relatively concise.

The rest of this paper is organized as follows. Section 2 reviews and analyzes existing related work. Section 3 introduces the basic data structures of blockchains and presents the general procedure framework of the *BCPs*. Section 4 formally presents the BaaV framework, elaborating on its inspiration and design details. Section 5 analyzes the BaaV framework from three perspectives: comparisons with multi-model work, security guarantees, and performance. Section 6 conducts experimental evaluations of BaaV. It tests the performance of several classic voting-based *BCPs* extended by the BaaV framework and examines the impact of network fluctuations on BaaV. Finally, the conclusions are presented in Section 7.

2. Related Work

The *BCPs* can be categorized into single-model and multi-model based on whether nodes can generate and confirm blocks in batches within each period.

2.1. Single-Model *BCPs*

The single-model *BCPs* can only confirm one block in each period.

The Nakamoto Consensus (NC) adopted by Bitcoin [3] was the first *BCP* for blockchains and is also a single-model *BCP*. In Bitcoin's setting, nodes generate a block roughly every ten minutes via the proof-of-work (PoW) mechanism, and the chain formed by these blocks grows gradually. A block is considered confirmed when the subchain consisting of its subsequent blocks reaches a length of six. Ignoring implementation-level optimizations (e.g., repeatedly executing block confirmation operations within a short time window), NC can confirm only one block per ten-minute period.

The core idea of NC is to characterize nodes' acceptance of a block by the length of its subchain. This block confirmation method is known as the longest chain rule. Under the longest chain rule, the block itself is the confirmation message. The longer the subchain of a given block, the more nodes in the network accept this block, and the higher the probability that the block will be confirmed. Existing works [13,14] have analyzed its

security. Some efforts [4,5] have improved NC. Ouroboros [5] modifies the PoW mechanism of NC, shortens the block generation interval, and retains the longest chain rule for block confirmation to guarantee security. The GHOST protocol [4] refines the longest chain rule into the heaviest subtree rule. Since a block may generate multiple forks among its descendant blocks, GHOST measures network nodes' acceptance of a block by the total number of blocks in these forks and accelerates block confirmation.

Another category of single-model *BCPs* generates additional voting information to complete block confirmation. These voting-based *BCPs* require nodes to vote for unconfirmed blocks and determine if a block can be safely confirmed based on the received votes. Their primary optimization direction lies in reducing the number of votes needed for confirming a block or the communication cost to broadcast the vote messages. The Byzantine fault tolerance (BFT) protocol can be used as a typical building block of voting-based *BCPs*. However, it requires multiple rounds of voting to confirm a block, which can result in significant communication costs. Some efforts [7,15] aim to reduce the communication cost through signature aggregation techniques. ByzCoin [11] modifies the block confirmation logic of NC. It selects a subset of nodes that have recently generated blocks to form a committee, which votes on unconfirmed blocks to achieve block confirmation. HotStuff [8,16] distributes the communication cost of BFT multi-round voting across different blocks, enabling the reuse of votes. Algorand [6] selects a small set of nodes to run a BFT protocol to reduce the communication cost. Other works design new confirmation mechanisms. Laska [9] and Self-Healing [12] randomly select nodes to vote for blocks and confirm a block if it is agreed on by over half of the nodes in the whole network with a large probability. VWeiST [10] works the same as Laska [9], except that it uses these votes as block weights and confirms a block if its weight cannot be surpassed by other competitors with a large probability.

Overall, these single-model *BCPs* feature simple designs and can provide rigorous security guarantees. Existing work [13,14] has proven that for the longest chain rule, the probability required for nodes to confirm the same block rises exponentially as the length of subsequent subchains increases. Hence, high probabilistic guarantees for distributed consistency can be achieved through appropriate parameter configuration. Meanwhile, studies on voting-based *BCPs* also provide their respective security proofs. Their approaches basically rely on random voting to estimate the overall network nodes' acceptance of blocks. According to the law of large numbers, as the number of votes increases, the voting result will reflect the overall acceptance of the block by all nodes.

However, since only one block can be confirmed in each period, these single-model *BCPs* suffer from low resource utilization of nodes and achieve poor *TPS*. Taking the amount of data processed per unit time as the metric for *TPS*, NC only reaches about 0.28 KB/s. Even voting-based *BCPs* can achieve merely around 2 MB/s with properly configured block sizes [10].

2.2. Multi-Model *BCPs*

Multi-model *BCPs* allow nodes to generate and confirm blocks in batches within one period. Sharding and directed acyclic graph (DAG) chains both employ multi-model *BCPs*.

Sharding partitions nodes into different groups to enable concurrent block processing. The *BCPs* in sharding can be understood as each group running a single-model *BCP* to maintain its own chain and collectively serve as a data layer to support upper-layer applications. Since multiple groups process blocks simultaneously and each group contains fewer nodes, the overall system performance is improved. However, the design of sharding *BCPs* is much more complex.

Firstly, sharding must ensure group security. Since each group runs a single-model *BCP* internally, sharding must ensure that the proportion of malicious nodes within each

group is below the protocol's fault tolerance threshold, otherwise the security of the entire system is compromised. One idea [17] to address group security is for each group of nodes to verify blocks both within its own group and in other groups. This means that groups will exchange information between each other, which might affect overall performance. Other efforts [18–21] randomly partition nodes into groups and subsequently regroup them. As long as the number of nodes in each group is sufficiently large, f in each group will converge to the global f . Even so, ensuring the security of all groups remains relatively complex and requires a trade-off between group size and security. Moreover, when detecting insecure groups or Sybil attacks, such methods typically need to regroup nodes. Regrouping nodes involves significant data exchange between nodes and is consequently notably slow.

Secondly, sharding needs to address the cross-shard transactions: when a transaction accesses data across different groups, how to ensure its atomic execution. Processing cross-shard transactions generally requires auxiliary information exchange across groups, e.g., the lock-based approach [19] and two-phase commit approach [17,18,22,23]. On this basis, some works have optimized communication and grouping to reduce communication cost [21,22,24,25].

In comparison, many single-model BCPs can easily provide clear security guarantees, e.g., ensuring the probability that all honest nodes confirm the same blocks is more than $1 - 10^{-15}$ [6,9,10,12]. However, it is difficult for the sharding BCPs to provide such precise descriptions of security. It should also be noted that some sharding BCPs make additional assumptions about the behavior of malicious nodes [24] or directly assume group security [22], which we consider unreasonable. Furthermore, a feature of some current sharding research is that it focuses on optimizing a single component without considering the complete consensus process. For example, some studies aim to reduce cross-shard transactions with limited discussion on group security [26,27].

DAG chains [28–30] allow a block to link multiple previous blocks, transforming the chain structure to a directed acyclic graph (DAG). In a DAG chain, when a block is confirmed, multiple previous blocks can be confirmed simultaneously. Unlike sharding, all nodes in DAG chains maintain a unified ledger. Therefore, they do not need to address the issues of group security and cross-shard transactions.

Current research on DAG chains' BCPs can be regarded as migrating existing protocols to the DAG setting, with some works introducing corresponding improvements during this process. Inclusive [30] directly applies NC to DAG chains and modifies its PoW mechanism to support the generation of more blocks within a single epoch, while still adopting the longest chain rule for block confirmation. Phantom [29] revises the confirmation mechanism, decomposing the DAG chains into distinct connected subgraphs and confirming blocks residing in multiple subgraphs. Spectre [31] can be viewed as the GHOST protocol adapted to the DAG setting. It assigns weights to preceding blocks based on votes cast by their successor blocks to evaluate the safety of those preceding blocks. When successor blocks link to multiple predecessors, weights are only assigned to the predecessor with the maximum weight. DAG-Rider [32] proposes a paradigm for constructing DAG chains, which split DAG construction into two modules: a reliable communication module and an ordering module. The former guarantees that all nodes maintain identical DAG views, whereas the latter performs deterministic ordering on data within the DAG view to ensure consensus among all nodes after data commitment. Essentially, the reliable communication module corresponds to the BCP of the DAG chain. Inspired by DAG-Rider, Bullshark [33] and Narwhal [34] realize block confirmation by introducing anchors into the DAG structure. In Bullshark, nodes periodically generate anchor blocks. An anchor block references a DAG graph via multiple preceding blocks, and once the anchor block is confirmed, the corresponding blocks within this DAG graph are committed simultaneously. Narwhal

decouples node functionalities and deploys validator nodes to verify DAG construction. Validators identify anchor blocks from the DAG chain to complete block confirmation.

In *BCPs*, when generating confirmation messages, nodes need to traverse from the last confirmed block to reach a leaf block. In the DAG chain, the successor blocks are logically equivalent; as a result, when nodes generate confirmation messages, the number of forks among unconfirmed blocks becomes exceedingly large, causing confirmation messages to be dispersed across different forks. This leads to persistent forks in the DAG chains. Given the fork issues, existing DAG chains are concentrated in the cryptocurrency domain, where transactions serve as the basic unit of data processing. Since transaction data are lightweight and propagate faster, they help reduce the number of forks. In addition, in the cryptocurrency domain, upper-layer requests must perform ownership verification when accessing data items, which can reduce the volume of conflicting transactions. Although DAG chains operate concurrently and with high resource utilization, they suffer from complex structures, persistent forks, and limited application scenarios.

Overall, although the multi-model *BCPs* achieve higher node resource utilization, they involve more sophisticated designs and pose greater difficulties in security analysis, which is critical for blockchain systems.

3. Background on Blockchain Consensus Protocol

3.1. Data Structure and Performance Metrics

In blockchain, the general format of a block is as shown in Equation (1). It consists of block header H and transaction set Tx . The header H contains the previous block's hash $Prev$, the public key pk , and the signature σ of the block-generator. $Prev$ is calculated by hashing the entire previous block, forming the chain structure between blocks. The transaction set Tx contains the upper-layer requests and data. For clarity of description, we introduce a *Height* field into the block header to denote the block's position within the chain structure. The height of the genesis block is 0, and the height of each subsequent block is derived by incrementing the previous block's height by one.

$$B = (H, Tx, \dots), H = (Prev, Height, \sigma, pk, \dots) \quad (1)$$

This chain structure exhibits transitivity: a node agrees on all preceding blocks if it agrees on a block, and any block is supported by all its subsequent blocks and their votes. As shown in Figure 1b, if a node agrees on B_3 , it also agrees on B_1 and B_2 , and B_1 is supported by the blocks B_2, B_3, B_4 and the votes $[v]_2, [v]_3, [v]_4$. Generally, a blockchain system also maintains a state database to record the state of all data items in the blocks. When a block is confirmed, nodes update the state database based on the Tx within this block. To avoid confusion, we use the term *ledger* to refer to the data layer of the blockchain system.

It should be noted that Equation (1) represents a relatively basic and general block structure. In some specific blockchain systems and applications, Equation (1) usually contains more fields. For example, in Ethereum [35–37], Tx fields are organized in the form of a Merkle tree, and the tree root is stored in the block header. The block header also contains many fields related to consensus protocol settings, such as the mining difficulty.

TPS and *Latency* are commonly adopted to measure the performance of blockchain systems. Generally, the calculation of *TPS* counts the number of data items contained in blocks. The result is dependent on the application-layer data format (i.e., Tx). Given the same block size, different application-layer data formats may lead to substantial differences in the number of data items that can be packed into a block. Although this paper implements a concrete transaction format, our primary focus lies on the design of the consensus protocol. Therefore, when calculating *TPS*, we evaluate the throughput using the size of Tx

within blocks, measured in MB/s. This measurement method has been adopted by many existing studies [6,9,10].

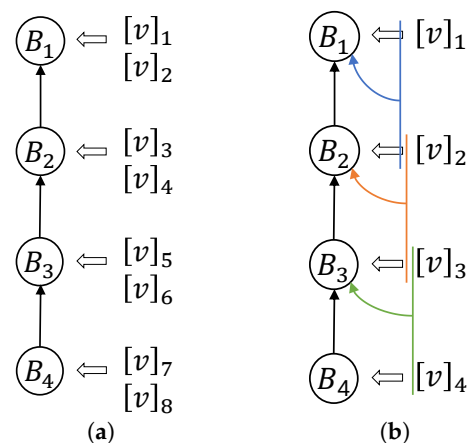


Figure 1. The example of the process of the BFT protocol and the BCP. Circles represent blocks, and $[v]_*$ represents one round of votes in the whole process, e.g., $[v]_1$ represents the first round of votes, $[v]_2$ represents the second round of votes. (a) The process of vanilla BCP; (b) the process of improved BCP.

3.2. Voting-Based Single-Model Blockchain Consensus Protocol

For the purpose of better understanding the execution process and design rationale of the voting-based BCPs, we first revisit the traditional Byzantine-fault-tolerant (BFT) protocols [2,8,38], which can be regarded as the building block of BCPs.

Using the simple (0,1) consensus problem as an example, the BFT protocols solve the problem by having nodes vote for either 0 or 1. When a majority of nodes prefer 0 or 1, each node transitions its state to the corresponding value. However, due to the presence of malicious nodes, these nodes may send divergent votes to different nodes, leading to inconsistencies in the final states across nodes. To tolerate such malicious behavior, the BFT protocols require multiple rounds of voting. In the context of blockchain, the values 0 and 1 correspond to a node's preference (agree or disagree) on a given block.

Combined with a block generation mechanism, a BFT protocol can construct a vanilla BCP. Its execution process can be as shown in Figure 1a. At any height in the chain structure, a node first generates a block, then all nodes run the BFT protocol for voting (assuming that the BFT needs two rounds of voting for block confirmation) and block confirmation. This BCP can work properly, but its trivial construction does not leverage the transitivity of the chain structure. Under this transitivity, $[v]_3, [v]_4$ can contain the agreement for B_1 and $[v]_1, [v]_2$. This implies that the multiple rounds of voting can be distributed across blocks at different heights, enabling the reuse of votes. The process of an improved BCP [8] is shown in Figure 1b. In the improved version, all nodes only need to cast a single round of voting for each block, and each node capitalizes on the transitivity to achieve confirmation. As shown, $[v]_1$ and $[v]_2$ can jointly confirm B_1 , while $[v]_2$ and $[v]_3$ can confirm B_2 .

The improved BCP can be used in some practical scenarios. However, several aspects still require further optimization. The most critical point is that each round of voting requires participation from all nodes. When the node scale is large, this inevitably impacts performance. Many studies [9,10,12] have discarded such BFT-style fault tolerance methods. Instead, they directly estimate the overall preference for blocks by randomly selecting nodes to vote. In comparison, this method requires fewer participating nodes per voting round, and leverages the transitivity of the chain structure to accumulate sufficient votes for block confirmation.

In fact, after this pattern of improved BCPs, almost all voting-based BCPs operate in a very similar manner. They periodically repeat the operations of block generation and voting. However, each block is not necessarily confirmed immediately after the voting operation of each period ends. Instead, the blocks will be confirmed until subsequent periods generate a sufficient number of votes.

Based on the above review, a voting-based single-model BCP can be generally represented by Algorithm 1. Within the *MAINPROC* of Algorithm 1, each node periodically repeats the *Generating block* (lines 6–8), *Voting* (lines 10–11), and *Confirming* (lines 13–15) steps. These three steps respectively define which nodes are able to generate blocks, which nodes are authorized to vote, and the criterion for determining when a block can be safely confirmed. Every Δ time, each node first determines whether it is a block-generator (line 6). If so, it calls *MainChain* to select a block at the end of the chain structure in *ledger* as the previous block (line 7), and generates a new block (line 8). Then, each node determines whether it is a voting node (line 10). If so, it votes for unconfirmed blocks (line 11). Finally, each node checks whether the received votes are enough to confirm a block (line 13) and updates *ledger* based on the *Tx* in this block (line 15). As this procedure proceeds, the vote messages received by each node gradually increase. Leveraging the transitivity, these accumulated votes eventually become sufficient to achieve block confirmation. In the *Confirming* step, a node can only confirm a single block each time. The *Confirming* step can be further subdivided into two sub-steps: *Making Decision* and *Committing*. The former determines whether blocks are available for confirmation, while the latter updates the latest state of the entire data layer according to the *Tx* fields within the confirmable blocks. The *Making decision* sub-step is critical, as it must guarantee that all honest nodes reach the same block decision. In addition, nodes must validate all received blocks and votes, including verifying the identity of their generators, which can be shown in *RECV* in Algorithm 1. It will be called to verify blocks or votes every time a node receives them and to add the valid votes or blocks to the *ledger* (lines 20–24).

Algorithm 1 Voting-Based Single-Model BCP Framework

```

1: Global Variables:
2:  $\Delta$ : the time interval of each period
3: ledger: the blockchain data layer
4: procedure MAINPROC
5:   while TimeUp( $\Delta$ ) do
6:     if isBlockGenerator() then                                ▷ Generating Block
7:       prev := MainChain(ledger)
8:       B = GenerateBlock(prev), Broadcast(B)
9:     end if
10:    if isVoter() then                                          ▷ Voting
11:      v = Vote(ledger), Broadcast(v)
12:    end if
13:    tB = MakeDecision(ledger)                                  ▷ Confirming:Making Decision
14:    if tB ≠ Null then
15:      Commit(tB)                                              ▷ Confirming:Committing
16:    end if
17:  end while
18: end procedure
19: procedure RECV(B,V)                                         ▷ Verifying block and vote
20:   if Verify(B) == true then
21:     AddToLedger(B)
22:   end if
23:   if Verify(V) == true then
24:     AddToLedger(V)
25:   end if
26: end procedure
  
```

Figure 2 shows the changes of the chain structure in an arbitrary node during the process of Algorithm 1. We use Laska [9] as an example for further elaboration. Laska proves that when over $1/2(1-f)N + fN$ nodes agree on a given block, after one round of voting from all nodes, any honest node will receive votes, a majority of which support this block. In Laska, a single node is randomly selected in each period to generate a block, followed by the random selection of n nodes to vote on unconfirmed blocks. As the process proceeds, in each node, the chain structure extends continuously, and votes are progressively received. During this process, multiple blocks may appear at the same height due to a block's failure to arrive on time, such as B_2 and B_3 at period t_3 . Despite the fork, the votes on B_2 and B_3 remain transitive and thus support B_1 . Throughout, nodes leverage these votes to compute the probability of the condition (over $1/2(1-f)N + fN$ nodes agree on a block) being satisfied. At period t_3 , the node uses votes v_a, v_b, v_c, v_d, v_e , and v_f to calculate that this condition for block B_1 is satisfied with a probability over $1 - \tau$ and thus confirms B_1 . Laska's block confirmation can be viewed as randomly sampling nodes to estimate the opinion of the overall nodes on a block. By the law of large numbers, as voting progresses, the final votes' results on a block probabilistically converge to all of the nodes' opinion on this block. Thus, Laska guarantees consistency with a precise probability.

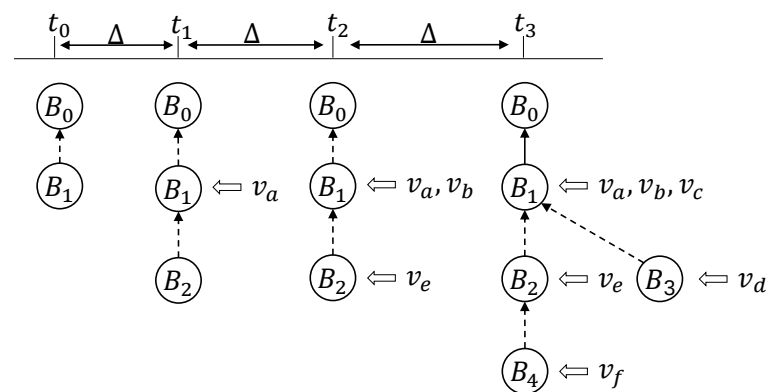


Figure 2. The process of a voting-based single-model BCP over time. Dashed arrows indicate that blocks have not yet been confirmed, and solid arrows indicate that blocks have been confirmed. For example, by period t_3 , B_1 has been confirmed, but B_2 has not yet been confirmed.

To ensure the security and effectiveness of a blockchain system, the BCPs need to possess consistency and liveness properties. The consistency and liveness properties can be defined as:

- *Probabilistic Consistency:* At any height in the ledger, all honest nodes confirm the same block with a probability over $1 - \tau$, where τ is a user-defined parameter.
- *Liveness:* As the blockchain system runs, nodes should be able to continuously confirm blocks.

It is noteworthy that some BFT-style BCPs [8] offer deterministic consistency. However, our description of probabilistic consistency is applicable to both cases. For BFT-style BCPs, $\tau = 0$, and for others, τ is typically set to a very small value.

4. Block-as-a-Vote Framework

This section presents a detailed introduction to the BaaV framework starting from its design inspiration, covering the basic data structures and various aspects of protocol modifications.

4.1. Inspiration

Our inspiration for the BaaV framework stems from two key observations.

First, the design pattern in voting-based *BCPs* follows the single model of “one node generates a block, followed by confirmation by all nodes.” This model results in very low utilization of nodes’ effective resources, leading to poor performance. Enhancing resource utilization necessitates concurrent block generation and batch confirmation. Notably, the processing of blocks and votes in such *BCPs* is inherently concurrent.

Secondly, in the voting-based *BCPs*, nodes essentially vote to select identical elements from a state set. Although each element is a block, nodes determine the security of an element based on the number of votes. This implies that the form of the element does not affect the final decision. Therefore, we can extend the element from a single block to multiple blocks. For instance, at period t_3 in Figure 2, B_1 has already been confirmed, and during the subsequent process, the state set becomes $\{B_2, B_3\}$. However, it is possible that all honest nodes have received v_a, v_b, v_c . Extending the set to $\{[B_2, v_a, v_b, v_c], [B_3, v_a, v_b, v_c]\}$ will have no impact on the subsequent *BCP* execution. Therefore, the voting-based *BCPs* possess inherent potential for concurrency and generalizability.

In this section, we introduce the block-as-a-vote (BaaV) framework. To scale the single-model voting-based *BCPs* to the multi-model, nodes must generate blocks in a batch and securely confirm blocks in a batch. The BaaV framework extends votes into blocks, effortlessly enabling batch generation of blocks. To achieve batch confirmation of blocks, the BaaV framework introduces an *Extend* field in its blocks to logically integrate multiple data-carrying items into a single unit. We first introduce the changes in data structure, and illustrate how the BaaV framework scales the voting-based *BCPs* using Figure 3 as an example; then, we discuss the modifications in *MAINPROC* and *RECV*.

4.2. Block and Chain Structure

The block format in the BaaV framework is shown in Equation (2). Compared to the original format shown in Equation (1), it has an additional *Extend* field and a *Type* field. The vote also adopts the format in Equation (2). Its *Prev* field indicates the block it supports. Its *Height*, σ , and *pk* fields only record auxiliary information, such as the public keys of voting nodes. They do not affect the logic of either the original protocol or the BaaV framework at all, so their internal values are ignored in the remaining discussion. BaaV’s extension to voting is limited to enabling it to carry data to realize concurrent block generation. Hence, its *Tx* field carries the upper-layer requests and data, but the *Extend* field is forever empty.

$$B = (H, Tx, Extend), H = (Prev, Height, Type, \sigma, pk) \quad (2)$$

$$Extend = [hash_i]_{i=1,2,\dots} \quad (3)$$

Since votes and blocks adopt the same data structure, their *Type* fields take different values to distinguish them. The BaaV framework still takes a block as the confirmation unit. To realize batch block confirmation, the *Extend* field of each block records the hash values of other blocks or votes. As is well known, a block itself is a logical integration of a large amount of upper-layer data. If a node votes for a block, it indicates that the node accepts the data within this block. The *Extend* field can be regarded as another form of data integration: it contains multiple blocks and votes by recording their hash values. If a node accepts this block, it also accepts the blocks and votes recorded in its *Extend* field. On this basis, when a node confirms a block, it will simultaneously confirm the blocks and votes contained in its *Extend* field to achieve concurrent confirmation.

After modifying the data structure, how can the BaaV framework achieve our goal? In the BaaV framework, the block-generating nodes add the hash of multiple blocks and votes to the *Extend* field of the new blocks, and when confirming a block, nodes will also confirm the corresponding blocks and votes in its *Extend* field. Figure 3 illustrates the

operational process of the BaaV framework. Considering the *Extend* field, the change of chain structure in an arbitrary node in the BaaV framework is illustrated in Figure 3a. In period t_1 , a node generates block B_2 and adds the hash of votes v_a and v_b to its *Extend* field. During the subsequent period from t_2 to t_{i+1} , all nodes repeat block generation and voting. For example, in period t_2 , a node generates B_3 and adds the hashes of v_c and v_d to its *Extend* field. Up to period t_{i+1} , the vote blocks v_c, v_d, v_e, v_f , and v_g are sufficient to confirm block B_2 . As a result, B_2 is confirmed, and v_a and v_b are also confirmed.

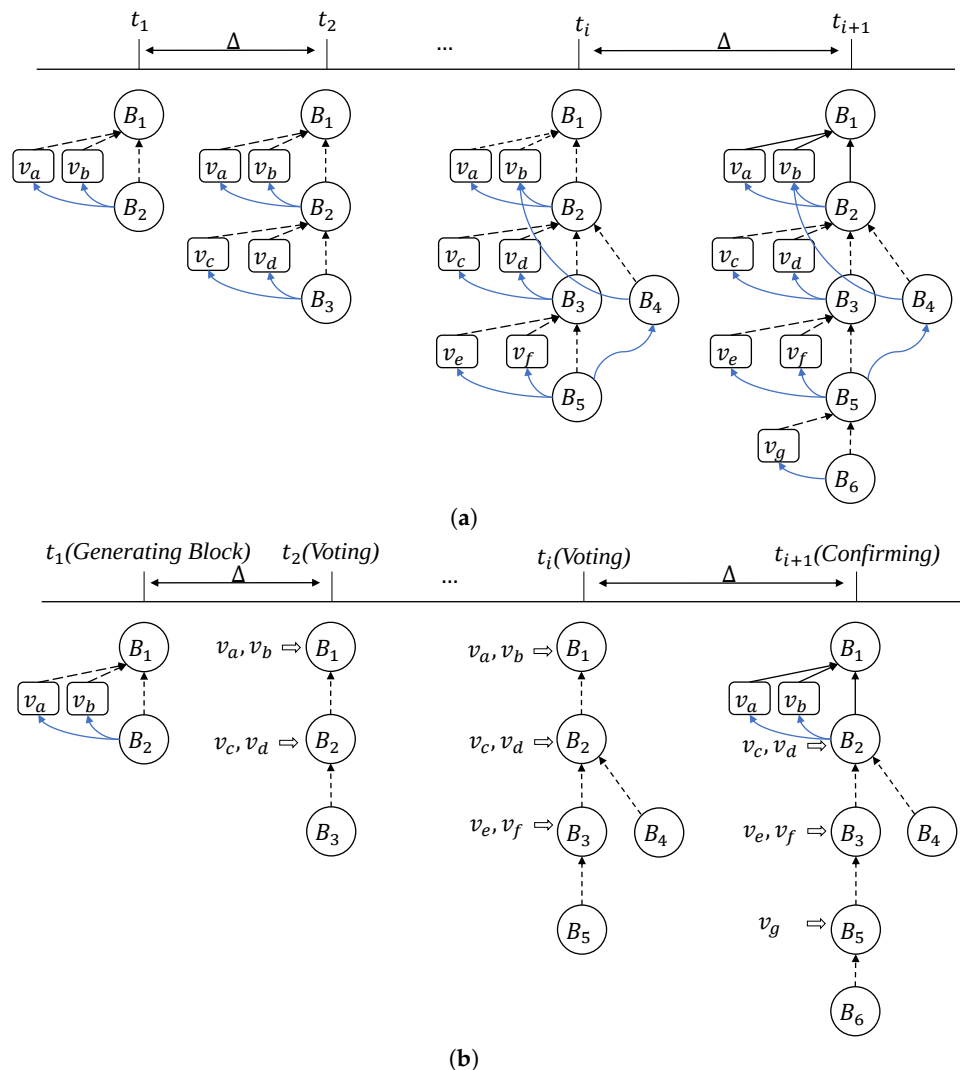


Figure 3. The process of the BaaV framework. The black arrows indicate the *Prev* relationship, the blue arrows indicate the relationship in *Extend*, and the square indicates that the votes carry data. (a) The change of ledger considering the *Extend* field; (b) details of different steps in BaaV.

Figure 3a illustrates the overall changes in the chain structure but does not capture the details of the *Voting* and *Confirming* in the BaaV framework. Figure 3b provides a more detailed view, using the confirmation of block B_2 as an example. Block B_2 requires voting across period t_2 to t_{i+1} to complete its confirmation. During voting in period t_2 to t_{i+1} , nodes first categorize blocks based on the *Type* field and ignore the *Extend* field in data blocks, and then strictly adhere to the original BCP in the *Voting* step. In the voting logic of the original protocol, nodes only care which block the vote supports and do not care whether it carries data and other fields. Therefore, in the voting step of Figure 3b, votes are not represented by a square. It can be observed that the *Voting* step in the BaaV framework is entirely identical to that of the original BCPs, with both performing the same voting

logic within the same chain structure. The *Confirming* step can be further divided into two sub-steps: *Making decision* (Algorithm 1 line 13) and *Committing* (Algorithm 1 line 15). The former uses voting messages to determine whether a block can be safely confirmed, while the latter updates the state database based on the block. In the *Making decision* sub-step of the BaaV framework, nodes similarly classify blocks and ignore the *Extend* field. When performing the final *Committing* sub-step, nodes simultaneously confirm blocks contained in the *Extend* field, i.e., v_a and v_b . Since v_a and v_b also carry data, the nodes achieve batch confirmation.

As shown in Figure 3b, the BaaV framework cleverly uses the *Extend* field to integrate multiple blocks into a single unit. Meanwhile, it utilizes the *Type* field to categorize blocks, thereby preserving the original chain structure and the original logic of the *Voting* and *Making decision* steps. Intuitively, the BaaV framework can inherit the security of the original BCPs.

4.3. Main Procedure

The BaaV framework is shown in Algorithm 2. Within the *MAINPROC* procedure, there are two differences from the previous version: in the *Generating Block* step, the *Extend* field needs to be completed, and in the *Committing* sub-step, blocks need to be committed in batches. The *Voting* and *Making Decision* steps remain basically unchanged.

Algorithm 2 BaaV Framework

```

1: Global Variables:
2:  $\Delta$ : the time interval of each period
3: ledger: the blockchain data layer
4:  $MB = \text{Map} < khash > < [vhash_i]_{i=1,...} >$                                 ▷ save the blocks with missing Extend blocks
5:  $tag = [hash_i]_{i=1,...}$                                                 ▷ indicate divergent blocks
6: procedure MAINPROC
7:   while TimeUp( $\Delta$ ) do
8:     treeview = IgnoreExtend(ledger)
9:     if isBlockGenerator() then                                       ▷ Generating Block
10:       Prev = MainChain(treeview)
11:       Extend = FindLeaf(ledger)
12:       B = GenerateBlock(Prev, Extend)
13:       Broadcast(B)
14:     end if
15:     if isVoter() then                                               ▷ Voting
16:        $v = \text{Vote}(\text{treeview}), \text{Broadcast}(v)$                          ▷  $v$ 's Tx field carries data
17:     end if
18:      $tB = \text{MakeDecision}(\text{treeview}, tag)$                              ▷ Confirming: Making Decision
19:     if  $tB \neq \text{Null}$  then
20:        $[B_i]_{i=1,...,l} = \text{FindExtend}(\text{ledger}, tB)$ 
21:        $\text{Commit}(tB, [B_i]_{i=1,...,l}, tag)$                                ▷ Confirming: Committing
22:     end if
23:   end while
24: end procedure
25: procedure RECV(B)                                                 ▷ Verifying
26:    $[B_i]_{i=1,...}, [mB_i]_{i=1,...} = \text{FindExtend}(\text{ledger}, B)$ 
27:    $prev = \text{FindPrev}(\text{ledger}, B.H.Prev)$ 
28:   if  $prev = \text{Null}$  then
29:      $[mB_i]_{i=1,...} = [[mB_i]_{i=1,...}, prev]$ 
30:   end if
31:   if VerifyData(B,  $[B_i]_{i=1,...}$ ) == false then
32:     Return                                                           ▷  $[B_i]_{i=1,...}$  is the received Extend blocks
33:   end if
34:   if Len( $[mB_i]_{i=1,...}$ ) == 0 then                                     ▷  $[mB_i]_{i=1,...}$  is the missing Extend blocks
35:     AddToChain(B)
36:   else
37:     UpdateMissing(B, MB,  $[mB_i]_{i=1,...}$ )
38:   end if
39:    $[dhash_i]_{i=1,...} = \text{DivergenceDetection}(B, \text{ledger})$ 
40:   UpdateTag( $tag, [dhash_i]_{i=1,...}$ )
41:    $[B_i]_{i=1,...} = \text{ReduceMissing}(B, MB)$ 
42:   AddToChain( $[B_i]_{i=1,...}$ )
43: end procedure

```

In the *MAINPROC* procedure, nodes first ignore the *Extend* field in the *ledger* to obtain the original chain structure, i.e., *treeview* (line 8). Before generating a block, the block-generator first obtains the *Prev* field in *treeview*, and traverses *ledger* to obtain all leaf blocks' hash as the *Extend*, and then generates a new block (lines 10–12). Adding all leaf blocks to the *Extend* field allows the block to integrate more blocks and votes for confirmation. The *Voting* and *Making Decision* steps are basically identical to before, except that votes carry data. Nodes perform voting and decision-making operations in the original chain structure, i.e., *treeview* (lines 15–18). In the *Committing* sub-step, if a block, denoted as *tb*, can be confirmed, nodes will start from *tb* and traverse the *ledger* toward the root, and then confirm all unconfirmed blocks and votes (lines 20–21) by hash order.

4.4. RECV Procedure

In the BaaV framework, the verification of blocks is more complex than before. The vote only appends data items, thus requiring only additional verification of these items. The blocks integrate multiple blocks and votes into a single unit via the *Extend* field, and their processing must be atomic, that is, either all blocks and votes in the *Extend* field are added to the chain structure or none are. It should be noted that since votes and blocks share the same data structure, with a slight abuse of notation, the pseudocode of the *RECV* procedure uses *B* to denote both of them. Considering that the *Extend* field of votes is empty, the additional verification of the *Extend* field of blocks imposes no extra impact on votes.

The verification of votes is straightforward. A node only needs to verify whether the node that generates the voting message has voting rights and whether the data carried in its *Tx* field are correct. This part of the logic is encapsulated in *VerifyData* (line 31).

When verifying a block *B*, a node may face two issues: first, due to network delays, some blocks or votes in *B*'s *Extend* field, named the missing items, may be still in transmission; second, *B* or its *Extend* field's items exhibit divergence.

For the missing items situation, nodes only need to wait to receive them to complete the verification. Each node maintains a map *MB* (line 4) that stores the hash of blocks (*khash*) and their missing *Extend* field's content ($[vhash_i]_{i=1,...}$).

When a malicious node is selected as a block-generator or voter, it can generate divergence blocks or vote and broadcast them to different nodes. For the divergence situation, each node maintains an array *tag* (line 5) to record all divergent blocks and votes, and only confirms the non-divergent part. Note that all *BCPs* inherently possess fault tolerance for divergent blocks, and some *BCPs* even leverage these divergent votes to accelerate block confirmation [10]. Therefore, the BaaV framework does not process these blocks; instead, it marks them in *tag*, allowing the original *BCP* logic to handle them (line 18).

In addition, *RECV* validates the data part of a block as well. If a block contains invalid transactions, it will be directly discarded (lines 31–32).

In the modified *RECV* procedure, when a node receives a block *B*, it first calls *FindExtend* to find its *Extend* field's item $[B_i]_{i=1,...}$ and missing item $[mB_i]_{i=1,...}$ (line 26). The node will verify all *Tx* fields of *B* and $[B_i]_{i=1,...}$, and discard this block if there are errors (lines 31–32). Then, the node will check if all the *Extend* field's items have been received. If so, it will add *B* to the *ledger*; otherwise, it will update the map *MB* (lines 34–37). Finally, the node detects if there are divergences in the *ledger* and updates the detection results (lines 39–40). In addition, block *B* may also be a missing item for other blocks, so nodes need to check whether any pending blocks in *MB* can be added to the *ledger* (lines 41–42).

Additionally, when any block lacks its previous block, it needs to be cached and waits for the previous block. The *RECV* procedure integrates the handling of missing previous block with the processing of *Extend* blocks (lines 27–29).

4.5. Committing Sub-Step

As analyzed in Section 4.4, the *Extend* field may contain divergent blocks. In addition, since upper-layer data are broadcast across the network, the blocks within *Extend* may also include duplicate or conflicting transactions, such as two transactions modifying the same data item. The *Committing* sub-step requires a more detailed explanation.

We use B_4 in Figure 4 to illustrate how to handle transaction-level conflicts and duplicates, and B_6 to demonstrate how to handle block-level divergence. When a node confirms data block B_4 , it needs to confirm both B_4 and $v_{b,1}$. At this point, no divergence is detected between B_4 , $v_{b,1}$, and preceding blocks. The node first sorts B_4 and $v_{b,1}$ by hash value. Assuming the sorted order is B_4 and $v_{b,1}$, nodes first update the state database using the transactions in B_4 , and then update the state database using the transactions in $v_{b,1}$. If there are duplicate or conflicting transactions in $v_{b,1}$ compared to B_4 , the node will skip these transactions.

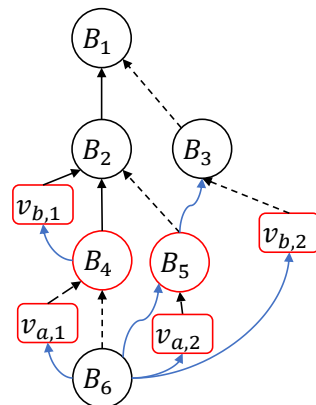


Figure 4. The *Committing* sub-step. The red indicates divergent blocks; $v_{a,1}$ and $v_{a,2}$ are two divergent vote from a same voter; $v_{b,1}$ and $v_{b,2}$ are two divergent votes from a same voter; B_4 and B_5 are two data blocks from the same block-generator.

As the *BCPs* continue to run, B_6 also meets the conditions for confirmation. Through traversing *ledger*, committing block B_6 requires simultaneously committing B_3 , B_5 , $v_{b,2}$, $v_{a,1}$, and $v_{a,2}$. Clearly, there are divergent blocks among them. The nodes first sort these blocks by hash value. Assuming the sorted order is B_6 , B_3 , B_5 , $v_{b,2}$, $v_{a,1}$, and $v_{a,2}$, nodes then commit the non-divergent parts in sequence. Since B_6 and B_3 are not divergent from any preceding blocks, the nodes first update the state database using their transactions. When committing $v_{b,2}$, it is divergent from the preceding committed block $v_{b,1}$, so it is skipped directly. When committing $v_{a,1}$, it is divergent from the next block $v_{a,2}$. However, since a single block $v_{a,1}$ or $v_{a,2}$ is not divergent from any others, the nodes commit $v_{a,1}$ and skip $v_{a,2}$.

From the case of block B_6 in Figure 4, it can also be observed that the *Extend* field may contain other blocks. The *FindExtend()* function may perform a layered search during its execution. This paper implements its search logic using a queue. Since this is a fundamental search operation on a graph-like data structure, the details are not elaborated here.

5. Analysis of BaaV Framework

This section compares the differences between the BaaV framework and DAG chains, and analyzes the security and performance aspects of BaaV.

5.1. Comparison with DAG Chains

In the example of Figure 3, the *Extend* field behaves very similarly to the DAG chain. This could lead to the misconception that the BaaV framework is merely a simple integration with the DAG chain. However, the BaaV framework differs significantly from the DAG chain in how concurrency is implemented and in their runtime behavior.

In terms of concurrency implementation, the idea of DAG chains is to allow multiple nodes to generate several blocks within the same period and achieve batch block confirmation via chain structures. From the design perspective, DAG chains introduce additional protocol complexity to equip single-model BCPs with concurrency. In contrast, the BaaV framework explores and extends the inherent concurrency of the original protocol. Therefore, the two designs start from fundamentally different motivations.

Regarding their runtime behavior, it should be emphasized that although both *Prev* and *Extend* contain the hash of blocks, but they serve different purposes. *Prev* maintains the original chain structure between blocks, allowing the original voting and confirmation mechanism to be used in the BaaV framework. *Extend* allows a block to integrate more blocks into a unit and is only used for confirming a batch of blocks. In terms of the critical logic of voting and decision making, the BaaV framework ignores *Extend* fields and still operates on the original chain structure.

In the DAG chains, the *Prev* field of a block is an array that saves the hash of multiple blocks. This means that all direct successors of a block are logically equivalent. Taking B_1 and B_2 in Figure 3 as an example, in the DAG, B_1 has three direct successor blocks: B_2 , v_a , and v_b . Due to communication delays, it is highly likely that nodes in the network have only received v_a , v_b , and B_2 separately. This means that all nodes actually need to vote to select one out of these three blocks. However, in the BaaV framework, B_1 has only one direct successor block B_2 , so all nodes only need to vote to select one block. Although v_a and v_b link to B_1 via their *Prev* fields, voting nodes can identify them as votes according to their *Type* field and thus will not cast votes for them.

It can be observed that the DAG structure leads to a significant increase in the number of unconfirmed blocks at each height. This feature disperses votes across different blocks at the same height, slowing down block confirmation. The BaaV framework, however, does not suffer from this issue.

5.2. Security Analysis

We first declare the key properties of the original BCPs: *Fault Tolerance* (Property 1) and *Object Generalizability* (Property 2).

Property 1 (Fault Tolerance). *In the presence of malicious nodes attempting to cast divergent votes or blocks, or other behavior, a voting-based BCP will eventually select honest nodes to produce new blocks during its process and ultimately confirm new blocks.*

Fault Tolerance is a fundamental property of all secure BCPs. It means that even in the presence of malicious nodes, the BCP can still securely generate and confirm blocks.

Voting-based BCPs essentially select one element from a set through voting, so their critical logic can be mathematically characterized. The element elected by voting is related to both the decision-making logic and the voting outcomes. Let function *MakingDecision* denote the mathematical logic of the decision-making logic. Within a BCP, *MakingDecision* is the mathematical representation of the internal logic of the *Making decision* sub-step (Algorithm 1 line 13). Let $[B_1^h, B_2^h, \dots, B_l^h]$ denote the blocks at any given height h , and $[v]^i$ the votes supporting B_i^h , and assume that the BCP determines B_i^h is confirmable. We have

$$\text{MakingDecision}(B_1^h, B_2^h, \dots, B_l^h, [v]^1, [v]^2, \dots, [v]^l) = B_l^h. \quad (4)$$

Object Generalizability (Property 2) means that under identical decision logic and voting outcomes, the form of elements in the set does not affect the decision result.

Property 2 (Object Generalizability). *From the perspective of all nodes, the blocks at any given height h in the ledger can be represented by the set $[B_1^h, B_2^h, \dots, B_l^h]$. After multiple rounds of voting, each block B_i^h is supported by votes $[v]^i$, and the voting-based BCP selects block B_l^h for confirmation based on the voting results. If each block B_i^h in the set is replaced with a different data structure E_i , the nodes will still be able to select the same data structure E_l from the set according to the same voting results. This property can be expressed by Equation (5).*

$$\begin{aligned} \text{If } \text{MakingDecision}(B_1^h, B_2^h, \dots, B_l^h, [v]^1, [v]^2, \dots, [v]^l) &= B_l^h \\ \text{MakingDecision}(E_1, E_2, \dots, E_l, [v]^1, [v]^2, \dots, [v]^l) &= E_l \end{aligned} \quad (5)$$

The difficulty in proving the security of the BaaV framework lies in two aspects. First, it is a framework that does not involve the specific internal logic of the BCP, which introduces a certain level of abstraction in the proof. For example, when a concrete BCP is not specified, the concrete form *MakingDecision* is unknown. Second, the *Extend* field may contain divergent blocks, which complicates the entire proof. We conduct proofs at two levels in this section. First, at the abstract level, we prove that as long as a BCP satisfies these two properties, the BaaV framework can inherit its security. At the concrete level, we take specific consensus protocols as examples to illustrate that they satisfy these two security properties and prove that they remain secure under the BaaV framework. Due to the different design logics and assumptions of various protocols, this section does not list all assumptions of these works. Instead, only the required assumptions are presented where needed in the proof, together with the corresponding explanations of the principles.

At the abstract level, as shown in Figure 4, the divergence with regard to the *Extend* field can be categorized into two scenarios: the *Extend* field itself may contain two divergent blocks, e.g., $v_{a,1}$ and $v_{a,2}$; or a block included in the *Extend* field may be divergent from subsequent blocks, e.g., B_4 and B_5 . Our proof approach is as follows: we demonstrate that for blocks at any given height, even if their *Extend* fields contain divergent blocks, confirmation can still be achieved after multiple rounds of voting. By recursively applying this conclusion starting from height 1, the proof can be completed. Since non-divergent blocks do not impact the proof, they will be omitted in the illustrative figure of the proof.

Our proof uses a common assumption that network messages between honest nodes are eventually deliverable. This is a fundamental assumption underlying almost all BCPs, because if messages between honest nodes cannot reach each other, network partitions may occur. According to the CAP theorem [39], in the presence of a partition, honest nodes in the network cannot simultaneously guarantee consistency and liveness.

Theorem 1. *If a voting-based BCP satisfies the properties of Object Generalizability and Fault Tolerance, the BaaV framework can inherit its Probabilistic Consistency and Liveness.*

Proof. Let the height of the last confirmed block be denoted as $h - 1$, where $h - 1$ is arbitrary. After a finite period of time, nodes in the network collectively generate some data block at height h , represented by the set $[B_1^h, B_2^h, \dots, B_l^h]$.

Considering the worst case (where malicious nodes do not generate data blocks and vote), as long as the BCP satisfies *Fault Tolerance*, at least one block generated by an honest

node will eventually be included in the set during its running process. Moreover, when no network partition occurs, this block and the *Extend*-related blocks will eventually be broadcasted to all honest nodes across the network. Therefore, under the BaaV framework, the *ledger* of all honest nodes will consistently extend with new blocks at its tail.

When $l > 2$, the consensus problem becomes multi-value. This scenario is taken into consideration in the design of every *BCP*, and a common approach is to vote for the block with the maximum hash value [9]. The value of l may affect the performance of the *BCPs*, but it does not compromise the security. For illustrative convenience, we assume $l = 2$ and use Figure 5 to demonstrate the *ledger* structure formed by B_1^h , B_2^h , and the blocks in their *Extend* fields.

It can be observed that overall B_1^h and B_2^h contain multiple pairs of divergent vote and data block, i.e., $(v_{b,1}, v_{b,2})$, $(v_{a,1}, v_{a,2})$, (B_1^h, B_3) . However, during subsequent periods of *Voting*, the BaaV framework ignores the *Extend* field and classifies the blocks via the *Type* field, as shown in the right part of Figure 5. Therefore, the entire process of *Voting* and the *Making Decision* step is entirely consistent with the original *BCPs*. The original *BCPs* are capable of tolerating such divergent scenarios, ensuring that all honest nodes select the same block for confirmation. Assuming B_1 is selected, and the *Extend* field extends B_1, B_2 into $[B_1^h, v_{a,1}, v_{b,1}, v_{b,2}]$, $[B_2^h, v_{b,2}, B_3]$, if the *BCP* satisfies *Object Generalizability*, these votes would similarly enable honest nodes to select $[B_1^h, v_{a,1}, v_{b,1}, v_{b,2}]$ for confirmation. During the *Committing* sub-step, the BaaV framework processes blocks in hash order, which is a deterministic operation. Consequently, all honest nodes can transition to the same data state.

Therefore, within the BaaV framework, nodes can generate and confirm identical blocks at any height of the *ledger*. By recursively applying this conclusion starting from height 1, the theorem holds. $\square$

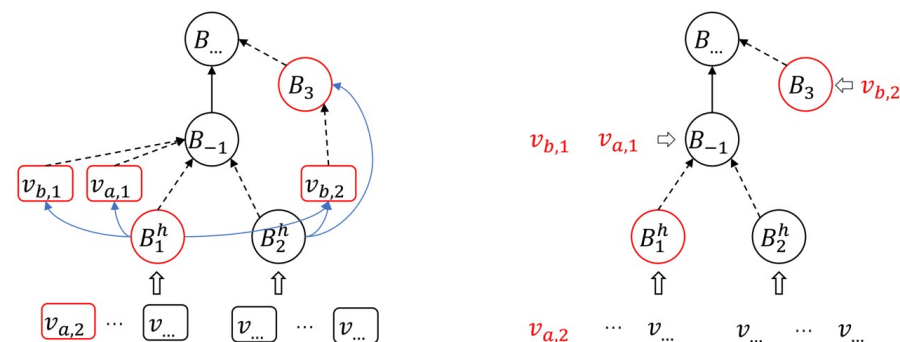


Figure 5. Illustration of BaaV's security.

At the concrete level, we select ByzCoin [11] and Laska [9] for the proof. We first provide a more detailed introduction to these two *BCPs*.

ByzCoin adopts the PoW mechanism of NC [3] to generate new blocks. It selects n nodes that generated the most recent blocks to vote on the pending blocks. A block is confirmed once it receives more than $2/3n$ votes. Essentially, ByzCoin picks an element from a set with maximum votes. Similarly, let the l blocks at height h be denoted as $[B_1^h, B_2^h, \dots, B_l^h]$, and the voting set of block B_i^h be $[v]^i$. The decision logic of ByzCoin can be expressed by Equation (6).

$$\text{MakingDesicion}(B_1^h, B_2^h, \dots, B_l^h, [v]^1, [v]^2, \dots, [v]^l) = B_i^h \quad \text{if} \quad |[v]^l| >= 2/3 * n \quad (6)$$

Its decision logic depends solely on the voting result. When this voting result is applied to any set of elements, Equation (7) still holds. Therefore, ByzCoin satisfies Property 2.

$$\text{MakingDesicion}(E_1, E_2, \dots, E_I, [v]^1, [v]^2, \dots, [v]^I) = E_I \quad \text{if} \quad |[v]^I| >= 2/3 * n \quad (7)$$

In Laska, one node is randomly selected in each period to generate a block. Then, some nodes are randomly selected to vote on the unconfirmed blocks. Based on vote results, Laska calculates whether the probability that more than half of the nodes prefer a given block exceeds $1 - \tau$. If so, the block is confirmed. Thus, the decision logic of Laska can be expressed by Equation (8):

$$\begin{aligned} \text{MakingDesicion}(B_1^h, B_2^h, \dots, B_I^h, [v]^1, [v]^2, \dots, [v]^I) &= B_I^h \\ \text{if} \quad \Pr(\sum_{i=1:N} \text{OPT}(i, I) > N/2 + \delta) &\geq 1 - \tau \\ \text{where} \quad \Pr(\sum_{i=1:N} \text{OPT}(i) > N/2 + \delta) &= \text{Stat}([v]^1, [v]^2, \dots, [v]^I) \end{aligned} \quad (8)$$

where δ is a constant correction term for handling malicious votes, $\text{OPT}(i, I) = 1$ if the i -th node in the network prefers the I -th block at height h and 0 otherwise, and Stat denotes a statistical estimation function used to compute the estimated probability of the target in $\Pr$. Stat is described in the original Laska paper [9]. It can be seen that although the mathematical form of Laska's decision logic is relatively complex, the final selection of B_I^h only depends on the voting result. If the blocks are replaced with arbitrary elements, Equation (9) still holds. Therefore, Laska satisfies Property 2.

$$\begin{aligned} \text{MakingDesicion}(E_1, E_2, \dots, E_I, [v]^1, [v]^2, \dots, [v]^I) &= E_I \\ \text{if} \quad \Pr(\sum_{i=1:N} \text{OPT}(i, I) > N/2 + \delta) &= 1 - \tau \\ \text{where} \quad \Pr(\sum_{i=1:N} \text{OPT}(i) > N/2 + \delta) &= \text{Stat}([v]^1, [v]^2, \dots, [v]^I) \end{aligned} \quad (9)$$

Theorem 2. Laska [9] satisfies the properties of Object Generalizability and Fault Tolerance, and the BaaV framework can inherit its Probabilistic Consistency and Liveness.

Proof. The proof in this part largely follows the original proof [9] and adopts its fundamental assumptions: messages between honest nodes are eventually deliverable, and there exists a global synchronous time (GST) in the network. After a GST, data among all honest nodes will be synchronized.

We first prove liveness. Assume that the height of blocks to be confirmed is h . After a GST, data among honest nodes will achieve temporary synchronization. Although each block is extended via the *Extend* field under the BaaV framework, the *RECV* procedure of BaaV (Algorithm 2 line 25) guarantees atomic block processing. Consequently, after the GST, all honest nodes hold identical blocks at height h , and they also receive blocks and votes associated with the *Extend* field. In subsequent voting rounds, nodes cast votes for these blocks either directly or transitively through chain structures. The tie can be resolved by comparing block hashes. During this process, most nodes vote for the same blocks. By the law of large numbers, as voting proceeds, voting results will converge in probability to the global outcome. Therefore, after some GSTs, the voting results enable the confirmation of blocks at height h .

We also use Figure 5 to assist in proving consistency. In Laska, if over $1/2(1 - f)N + fN$ nodes accept a block, then even in the presence of malicious voting nodes, after all nodes cast one round of votes, over half of the votes still support this block, so this block can be confirmed. Laska randomly selects nodes to cast votes and calculates the probability that

the above judgment condition holds via hypothesis testing based on voting results. In this proof, we refer to this condition as the decision condition. A block can be confirmed if the probability that the decision condition fails for this block is less than τ .

Assume the height of the blocks to be committed is h . The confirmation of these blocks only depends on subsequent *Voting* and *Making Decision* steps. In these steps, the BaaV framework reduces the ledger to the original chain structure, as shown on the right side of Figure 5. Laska adopts proof by contradiction. Assume that an honest node $node_1$ confirms block B_1^h , implying that the decision condition fails for B_1^h if it is less than τ based on the voting results. If another honest node $node_2$ confirms block B_2^h , this implies that the decision condition fails for B_2^h if it is less than τ based on the voting results. The probability that these two conditions fail simultaneously is less than τ^2 . Therefore, when $node_1$ calculates the failure probability of the decision condition, it can obtain at most τ^2 , which is greater than τ , so block confirmation cannot be achieved. Hence, probabilistic consistency holds. $\square$

5.3. Performance Analysis

Regarding the BaaV framework, some potential concerns may arise: By integrating multiple blocks into a unit, could their verification become a performance bottleneck? What impact would network delays have on it?

The verification of a data block and the blocks in its *Extend* field does not occur simultaneously. If a block-generator adds the hash of block B' to a new block B , this indicates that B and B' were generated in different periods. Consequently, nodes in the network receive and process these two blocks at different times. From this perspective, the BaaV framework distributes the data load across data blocks and vote, allowing nodes in the network to handle these loads in a time-shared manner. This reduces idle time among network nodes and mitigates the risk of any single node becoming a performance bottleneck.

The impact of network delay cannot be hastily determined. When network delay is high, blocks in the *Extend* field of a data block may require longer propagation times. If this data block's *Extend* field is large, it may remain in the cache for a long time. Intuitively, this could lead to performance degradation. However, we argue that the BaaV framework itself exhibits robustness against the impacts induced by network delay. When the network is unobstructed, the vote and data blocks generated in each period can be broadcast quickly and stably across the entire network. In such cases, even if a data block's *Extend* field contains numerous blocks, it can still be rapidly verified and added to the *ledger*. When the network is congested, block propagation slows down. Consequently, block-generator nodes will only add a small number of block hashes to the *Extend* field when generating data blocks. Hence, the verification waiting time for such blocks remains within an acceptable range. It can be observed that the size of the *Extend* field exhibits a positively correlated change with network conditions. The impact of network delay will be investigated through experiments.

If occasional network packet loss leads to the loss of block transmissions, some data blocks may be permanently saved in cache for missing the *Extend* field's block. However, the subsequent block-generator can still include the vote at the leaf of the *ledger* in their *Extend* fields. In the BaaV framework, only data blocks can be cached due to the processing of the *Extend* field. As a result, the cached data load caused by such occasional packet loss is limited, and its impact on performance is similarly constrained. Similar scenarios also exist in the single-model BCPs. When a block's previous block is missed due to the transmission, it remains cached even if it is broadcast across the network. A common solution for this problem is that when detecting a long-cached block, a block-generator will repackage its transactions into a new block.

6. Performance Evaluation

This section experimentally evaluates the performance of the BaaV framework.

6.1. Experimental Setting

We implement several blockchain consensus methods in Golang, containing about 8000 lines of core code. We separate the *Making Decision* and *Committing* steps and repeat them at a shorter interval Δ' to achieve better performance. We test the performance improvements the BaaV framework can bring under different data loads. Let N denote the number of whole nodes, and n the number of voting nodes in each period. For BCPs under BaaV, we set the data block size to 1MB and gradually increase the vote size sv from 100 KB to 1300 KB. For the original BCPs, the block size is set to $1\text{MB} + sv \times n$ to achieve the same data load. We measured the overall TPS (MB/s) on confirming 100 blocks and the average confirmation *Latency* for each block. We recorded the message delay and the time on unmarshalling blocks and votes to assess the workload of the network and CPU. In presenting the results, we use solid lines to represent the results under BaaV, while dashed lines denote the results for the original BCPs and other baselines. Some general settings are as follows: $\tau = 10^{-10}$ for probabilistic consistency, $f = 0.2$, $\Delta = 5$ s, $\Delta' = 2$ s, each node connects with five nodes with 50 Mbps bandwidth. Before communication, nodes randomly sample from an exponential distribution with a mean of 100ms as the network delay. The general settings are referenced from existing works and are assigned similar values. The malicious nodes, when selected as the block-generators, generate two blocks, and when chosen as the voting nodes, produce two divergent votes. Our experiments were conducted on four Ubuntu servers with the following hardware: CPU E5-2680, 126G memory, bandwidth 10,000 Mbps.

Baseline Setting. We select three state-of-the-art single-model BCPs with rigorous security analysis, HotStuff [8], Laska [9], and VWeiST [10], as baselines to evaluate their performance in our BaaV framework. We chose two rigorously analyzed multi-model BCPs, Monoxide [17] and RapidChain [18], for comparison. We combine the data grouping protocol of BrokerChain [26] with these two protocols to construct two sharding systems. Note that HotStuff uses the BFT for block confirmation and is typically not designed for large-scale nodes, whereas Monoxide and RapidChain target large-scale nodes. We omitted comparisons between HotStuff and Monoxide and RapidChain. Bullshark [33] is selected to construct the DAG chain. The DAG of Bullshark is built with blocks as the basic unit. Since Bullshark performs block confirmation based on the BFT protocol, the original literature does not describe how to extend it to large-scale networks. Therefore, we omit the comparison between Bullshark and Laska as well as VWeiST under large-scale node settings.

6.2. Small-Scale Nodes Scenario

We first test the performance of HotStuff [8] under the BaaV framework, named HotStuff-BaaV, when $N = n = 30$. The Pacemaker implementation (a module in HotStuff) is as follows: a node is randomly selected to generate a block in each period, and it must wait for the block from the previous period before generating a block.

As shown in Figure 6, both HotStuff and HotStuff-BaaV initially experienced a linear improvement in TPS before encountering a plateau. However, HotStuff reaches its TPS plateau much earlier ($sv = 300$ KB), whereas HotStuff-BaaV reaches its plateau when $sv = 1100$ KB. In terms of *Latency*, HotStuff deteriorates rapidly. However, HotStuff-BaaV shows minimal changes until $sv = 1300$ KB. Even so, the increase in *Latency* is very small compared to HotStuff.

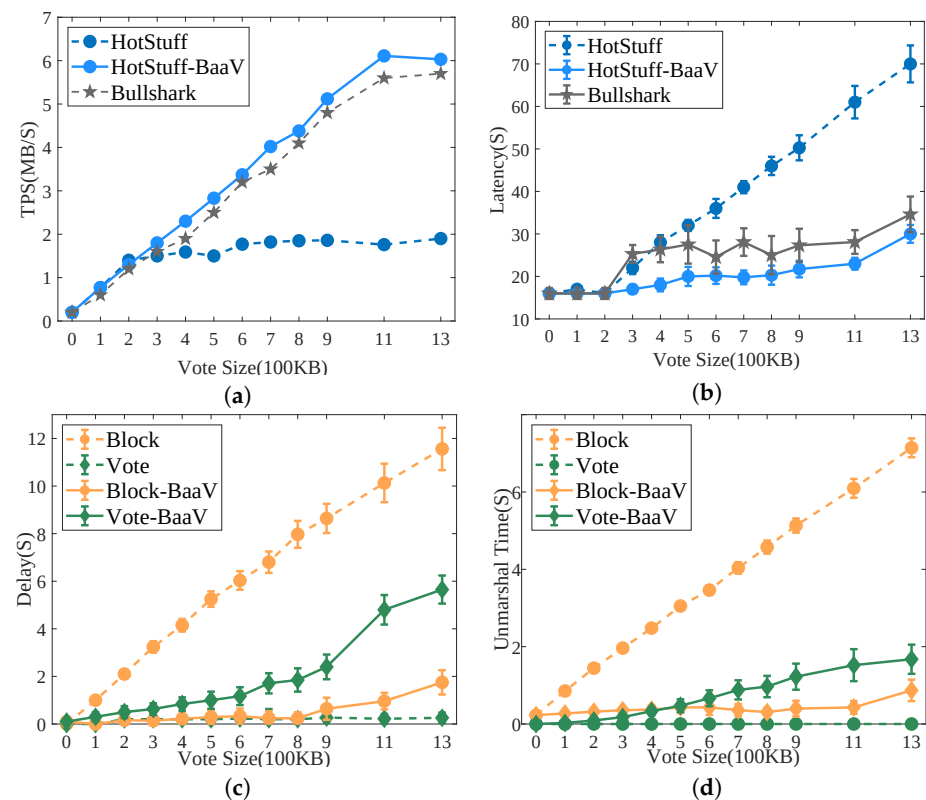


Figure 6. HotStuff vs. HotStuff-BaaV. The error bar is displayed as mean $\pm$ 1/2std. (a) TPS; (b) Latency; (c) Message Delay; (d) Unmarshal Time.

Figure 6c,d show the network load and CPU load of HotStuff and HotStuff-BaaV. In HotStuff, votes carry no data, resulting in significantly lower average propagation delay and unmarshal time. As shown, in HotStuff, the message delay and unmarshal time on block increase linearly with the data load and rapidly exceed Δ and Δ' . This may have a two-fold impact on performance: the new block-generator delays generating new blocks because they have not received blocks generated in the previous periods, or nodes have received new blocks but fail to unmarshal them within Δ' seconds, missing the current *Making Decision* and *Committing* steps, resulting in performance degradation. In contrast, HotStuff-BaaV exhibits significantly lower network and CPU loads. The primary reason is that the BaaV framework distributes the data load generated within a period across different nodes, which process the load in a concurrent and time-shared manner, resulting in highly efficient utilization of node resources. As shown, in HotStuff-BaaV, when $sv = 1300$ KB, the message delay and unmarshal time for vote nearly exceed Δ and Δ' , respectively. This means that the data load at this point exceeds the upper limit of the utilized resources, leading to a performance plateau. When $sv = 1300$ KB, the total data load generated per period is approximately 40MB, which is considered a substantial load in many common blockchain systems.

Compared with the DAG chain Bullshark [33], HotStuff-BaaV achieves a similar throughput performance. This is because the two protocols share comparable workflows, and the total volume of data generated and processed per unit time is identical. However, Bullshark needs to periodically generate anchor blocks to confirm and order the overall DAG, resulting in slightly lower measured throughput than HotStuff-BaaV. In terms of block confirmation latency, HotStuff-BaaV achieves distinctly better performance. The reason is that the DAG in Bullshark is unstructured. Multiple predecessors and successors of any block are logically equivalent. Therefore, from the perspective of chain structure, the DAG chain contains a large number of divergences. When the load rises moderately,

block propagation latency increases. This creates discrepancies in the DAG maintained by different nodes and further slows down block confirmation.

6.3. Large-Scale Nodes Scenario

We test Laska [9] and VWeiST [10] under the BaaV framework when $N = 2000$. Both of them select a small scale of nodes to vote in each period. Due to hardware limitations, we ran 200 nodes and set 30 nodes to vote ($n = 30$) in each period, and a random algorithm was used to determine whether a node was selected from honest or malicious nodes. For Monoxide and RapidChain, we also ran 200 nodes, averagely divided into 30 groups. The block size generated in each group was set to $(1\text{MB} + sv \times 30)/30$. Since Laska and VWeiST are identical when generating blocks and voting, we only sum the message delay and unmarshal time from the VWeiST experiments.

As shown in Figure 7, the TPS of VWeiST-BaaV and Laska-BaaV increases linearly with the data load and then reaches a plateau, while their Latency changes very slowly until $sv = 1300$ KB. However, the improvement of TPS on VWeiST and Laska is insignificant, and their Latency deteriorates faster. As shown, both VWeiST-BaaV and Laska-BaaV outperform Monoxide and RapidChain in terms of TPS and Latency. The main reason is that when handling cross-group transactions, some groups in Monoxide and RapidChain need to wait for other groups to complete block confirmation, which slows down block confirmation. Moreover, we observe that RapidChain's Latency is higher than Monoxide's, and its error margin is also larger, which is caused by RapidChain's regrouping operation. Similarly, as shown in Figure 7c,d, under the BaaV framework, the communication load and CPU load of nodes are significantly lower and change more slowly, which proves that our BaaV framework can effectively improve the utilization of node resources to achieve better performance.

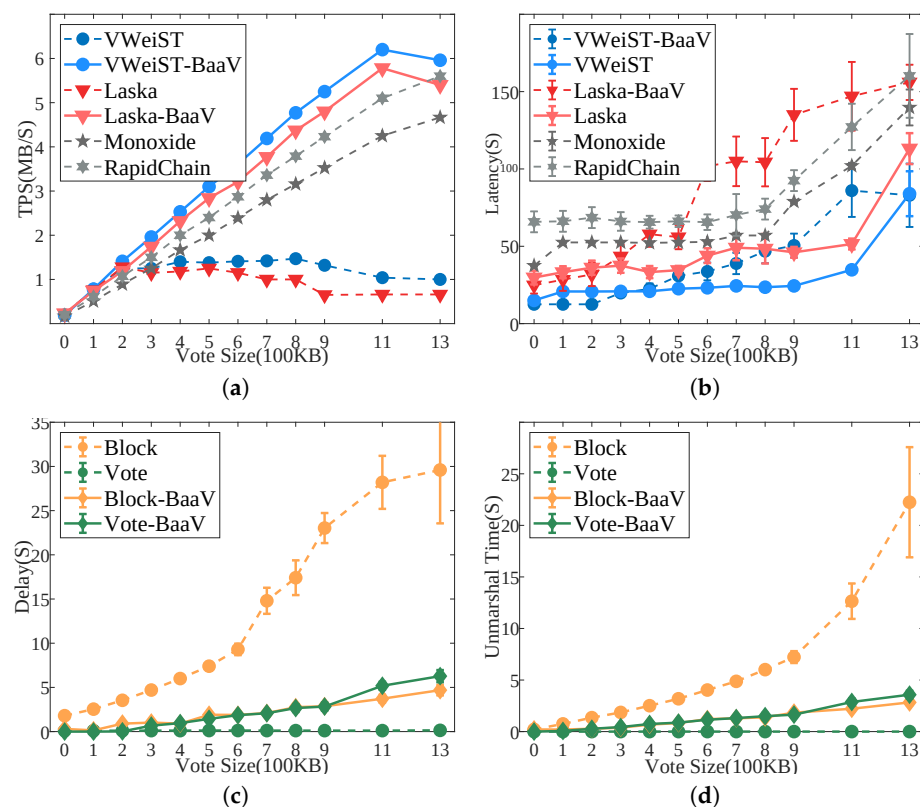


Figure 7. VWeiST, Laska vs VWeiST-BaaV, Laska-BaaV. The error bar is displayed as mean $\pm 1/2$ std. (a) TPS; (b) Latency; (c) Message Delay; (d) Unmarshal Time.

6.4. Network Fluctuation Scenario

To investigate the impact of network fluctuations on performance, we vary the network delay: each node samples from a uniform distribution over $[0, T]$ as the network delay before every communication. We select Laska [9] as an example to examine how BaaV's performance changes with network delay. We set sv to 700 KB, keep other parameters unchanged, and vary T from 1 s to 20 s. As shown in the Figure 7, a sv of 700 KB does not exceed the resource limits of the nodes, so we omit measurements of the workload of the network and CPU.

As shown in Figure 8, the BaaV framework demonstrates robustness against network fluctuations. When network delay does not exceed 15 s, i.e., $T < 15$ s, BaaV is able to maintain a decent level in both *TPS* and *Latency*. One observation from the *Latency* caught our attention: when $T < 15$ s, the *Latency* variance of the BaaV framework is significantly larger than that of Laska. By examining the experimental logs, we found that since the propagation time for both data blocks and vote increases, data blocks from multiple adjacent periods will appear at the same height. These blocks will gradually be directly or indirectly referenced by data blocks in subsequent periods through the *Extend* field. Therefore, their confirmation time is the same as that of blocks in the subsequent period, which explains why the variance in *Latency* is significantly large. When subsequent blocks reference a large number of data blocks generated in previous periods, their caching time increases. However, when they are confirmed, nodes will batch-confirm a large number of blocks in *Extend* fields, which explains why BaaV can maintain a commendable *TPS*. When the network delay exceeds the robustness limit of the BaaV framework, i.e., $T = 20$, its performance declines noticeably. Under the same conditions, Laska also performs poorly as well.

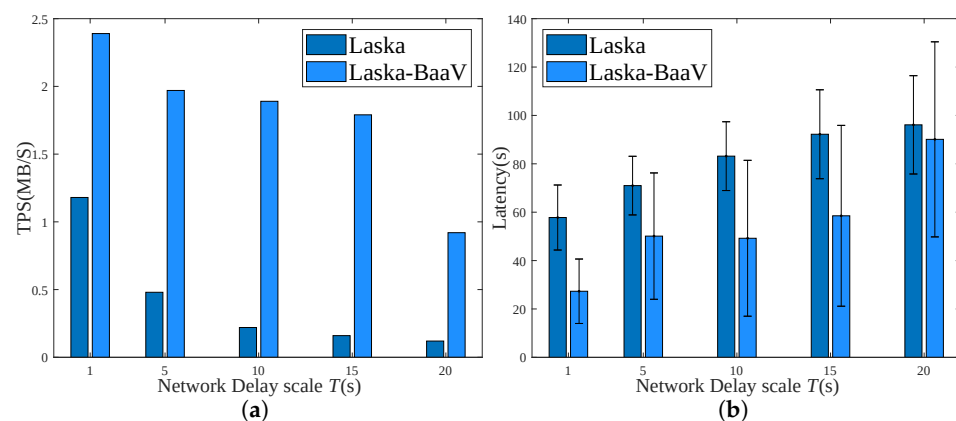


Figure 8. Laska vs Laska-BaaV in network fluctuation scenario. The error bar is displayed as mean $\pm 1/2$ std. (a) *TPS*; (b) *Latency*.

7. Conclusions

Blockchain has emerged as a novel paradigm for ensuring data trustworthiness; however, its inherent performance bottlenecks substantially constrain the scalability and wider adoption of its application scenarios. At present, blockchain technology is predominantly deployed in domains such as digital finance and data sharing. Nevertheless, owing to insufficient throughput, many blockchain system designs make concessions in decentralization and security in order to satisfy the requirements of upper-layer applications. For instance, blockchain systems built upon sharding or DAG chains encounter considerable difficulty in ensuring rigorous and unambiguous security guarantees. Conversely, certain systems—such as sharding [40,41] and Fabric [42]—introduce centralized components to achieve performance gains, which stands in direct contradiction to the decentralization prin-

ciple of blockchain itself. As can be seen, although these improvements boost performance, they make blockchain systems lose their original nature and become self-contradictory. This paper offers a lossless approach that preserves the essential nature of blockchain while simultaneously enhancing performance. Moreover, the proposed improvement to the consensus protocol does not alter its core logic and thus can be applied to existing blockchain systems [6,8] with a lightweight modification.

This paper investigates the low *TPS* problem of blockchain consensus protocols, reviews the design paradigms of existing blockchain consensus protocols, and explores the root causes of their poor performance. Depending on whether a protocol can generate and confirm blocks in batches, we classify such protocols into the single-model and the multi-model categories. The former can only confirm one block per period, resulting in low utilization of node resources. Nevertheless, it features a concise design and provides strong security guarantees. The latter supports batch block processing, yet it suffers from complicated system-wide design and makes security arguments more challenging. Through analysis, we find that voting-based protocols in the single-model version inherently contain concurrency, and their vote targets are generalizable.

Based on these two observations, we propose the block-as-a-vote (BaaV) framework, which can scale existing voting-based blockchain consensus. In the BaaV framework, blockchain consensus can concurrently generate multiple blocks and achieve batch confirmation of blocks while retaining the original security guarantees. The BaaV framework significantly improves the utilization of node resources, enabling *TPS* to increase linearly with data load when the load does not exceed the overall capacity. In our experimental tests, the BaaV framework can raise the upper bound of node *TPS* from approximately 2 MB to roughly 6 MB and maintain an almost unchanged *Latency*.

Author Contributions: Conceptualization, H.X. and J.L.; methodology, H.X. and J.L.; software, H.X.; validation, H.X. and J.L.; formal analysis, H.X.; investigation, H.X. and J.L.; resources, H.X. and J.L.; data curation, H.X. and J.L.; writing—original draft preparation, H.X.; writing—review and editing, H.X. and J.L.; visualization, H.X.; supervision, J.L.; project administration, J.L.; funding acquisition, J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Anhui University Blockchain Application Alliance and Strategic Priority Research Program (Category A) of the Chinese Academy of Sciences (No. XDA19020102).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author due to pending software copyright registration.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Habib, G.; Sharma, S.; Ibrahim, S.; Ahmad, I.; Qureshi, S.; Ishfaq, M. Blockchain Technology: Benefits, Challenges, Applications, and Integration of Blockchain Technology with Cloud Computing. *Future Internet* **2022**, *14*, 341. [CrossRef]
2. Lamport, L.; Shostak, R.E.; Pease, M.C. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* **1982**, *4*, 382–401. [CrossRef]
3. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Available online: <http://satoshinakamoto.me/bitcoin.pdf> (accessed on 7 October 2026).
4. Sompolinsky, Y.; Zohar, A. Secure High-Rate Transaction Processing in Bitcoin. In Proceedings of the International Conference on Financial Cryptography and Data Security, San Juan, Puerto Rico, 26–30 January 2015; pp. 507–527.
5. Kiayias, A.; Russell, A.; David, B.; Oliynykov, R. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *Proceedings of the 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, 20–24 August 2017*; Katz, J., Shacham, H., Eds.; Springer: Berlin/Heidelberg, Germany, 2017; pp. 357–388.

6. Gilad, Y.; Hemo, R.; Micali, S.; Vlachos, G.; Zeldovich, N. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28 October–1 November 2017; pp. 51–68.
7. Li, P.; Wang, G.; Chen, X.; Long, F.; Xu, W. Gosig: A scalable and high-performance byzantine consensus for consortium blockchains. In Proceedings of the 11th ACM Symposium on Cloud Computing, Online, 19–21 October 2020; pp. 223–237.
8. Yin, M.; Malkhi, D.; Reiter, M.K.; Gueta, G.G.; Abraham, I. HotStuff: BFT Consensus with Linearity and Responsiveness. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, Toronto, ON, Canada, 29 July–2 August 2019; pp. 347–356.
9. Reijnders, D.; Szalachowski, P.; Ke, J.; Li, Z.; Zhou, J. Laksa: A probabilistic proof-of-stake protocol. In Proceedings of the 28th Annual Network and Distributed System Security Symposium, Online, 21–25 February 2021.
10. Xiong, H.; Qu, C.; Li, J. VWeIST: A Scalable and Efficient Proof-of-Stake Blockchain Consensus. In Proceedings of the the 2024 ACM Symposium on Cloud Computing, Redmond, WA, USA, 20–22 November 2024; pp. 637–649.
11. Kokoris-Kogias, E.; Jovanovic, P.; Gailly, N.; Khoffi, I.; Gasser, L.; Ford, B. Enhancing bitcoin security and performance with strong consistency via collective signing. In Proceedings of the 25th USENIX Conference on Security Symposium, Austin, TX, USA, 10–12 August 2016; pp. 279–296.
12. Dou, H.; Yin, L.; Lu, Y.; Xu, J. A probabilistic Proof-of-Stake protocol with fast confirmation. *J. Inf. Secur. Appl.* **2022**, *68*, 103268. [[CrossRef](#)]
13. Pass, R.; Seeman, L.; Shelat, A. Analysis of the Blockchain Protocol in Asynchronous Networks. In Proceedings of the 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, 30 April–4 May 2017; pp. 643–673.
14. Garay, J.A.; Kiayias, A.; Leonardos, N. The Bitcoin Backbone Protocol: Analysis and Applications. *J. ACM* **2024**, *71*, 25. [[CrossRef](#)]
15. Golan Gueta, G.; Abraham, I.; Grossman, S.; Malkhi, D.; Pinkas, B.; Reiter, M.; Seredinschi, D.A.; Tamir, O.; Tomescu, A. SBFT: A Scalable and Decentralized Trust Infrastructure. In Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Portland, OR, USA, 24–27 August 2019; pp. 568–580.
16. Cheng, T.; Zhou, W.; Yao, S.; Feng, L.; He, J. Multi-pipeline HotStuff: A High Performance Consensus for Permissioned Blockchain. In Proceedings of the 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Wuhan, China, 20–22 March 2022; pp. 1008–1019.
17. Wang, J.; Wang, H. Monoxide: Scale out Blockchains with Asynchronous Consensus Zones. In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation, Boston, MA, USA, 26–28 February 2019; pp. 95–112.
18. Zamani, M.; Movahedi, M.; Raykova, M. RapidChain: Scaling Blockchain via Full Sharding. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 931–948.
19. Luu, L.; Narayanan, V.; Zheng, C.; Baweja, K.; Gilbert, S.; Saxena, P. A Secure Sharding Protocol For Open Blockchains. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October, 2016; pp. 17–30.
20. Li, M.; Lin, Y.; Zhang, J.; Wang, W. CoChain: High Concurrency Blockchain Sharding via Consensus on Consensus. In Proceedings of the IEEE INFOCOM 2023, New York City, NY, USA, 17–20 August 2023; pp. 1–10.
21. Kokoris-Kogias, E.; Jovanovic, P.; Gasser, L.; Gailly, N.; Syta, E.; Ford, B. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In Proceedings of the 2018 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 21–23 July 2018; pp. 583–598.
22. Tang, H.; Zhang, H.; Zhang, Z.; Jin, C.; Zhou, A. Towards High-performance Transactions via Hierarchical Blockchain Sharding. In Proceedings of the 30th European Conference on Parallel and Distributed Processing, Madrid, Spain, 26–30 August 2024; pp. 373–388.
23. Dang, H.; Dinh, T.T.A.; Loghin, D.; Chang, E.; Lin, Q.; Ooi, B.C. Towards Scaling Blockchain Systems via Sharding. In Proceedings of the 2019 International Conference on Management of Data, Amsterdam, The Netherlands, 30 June–5 July 2019; pp. 123–140.
24. Hong, Z.; Guo, S.; Li, P.; Chen, W. Pyramid: A Layered Sharding Blockchain System. In Proceedings of the 40th IEEE Conference on Computer Communications, Vancouver, BC, USA, 26–30 July 2021; pp. 1–10.
25. Amiri, M.J.; Lai, Z.; Patel, L.; Loo, B.T.; Lo, E.; Zhou, W. Saguaro: An Edge Computing-Enabled Hierarchical Permissioned Blockchain. In Proceedings of the 39th IEEE International Conference on Data Engineering, Anaheim, CA, USA, 26–29 July 2023; pp. 259–272.
26. Huang, H.; Peng, X.; Zhan, J.; Zhang, S.; Lin, Y.; Zheng, Z.; Guo, S. BrokerChain: A Cross-Shard Blockchain Protocol for Account/Balance-based State Sharding. In Proceedings of the IEEE INFOCOM 2022, London, UK, 20 June 2022; pp. 1968–1977.
27. Nguyen, L.N.; Nguyen, T.D.T.; Dinh, T.N.; Thai, M.T. OptChain: Optimal Transactions Placement for Scalable Blockchain Sharding. In Proceedings of the 39th IEEE International Conference on Distributed Computing Systems, Dallas, TX, USA, 7–10 June 2019; pp. 525–535.
28. Fan, C.; Ghaemi, S.; Khazaei, H.; Chen, Y.; Musilek, P. Performance Analysis of the IOTA DAG-Based Distributed Ledger. *ACM Trans. Model. Perform. Eval. Comput. Syst.* **2021**, *6*, 1–10. [[CrossRef](#)]

29. Sompolinsky, Y.; Zohar, A. PHANTOM GHOSTDAG: A scalable generalization of Nakamoto consensus. In Proceedings of the 3rd ACM Conference on Advances in Financial Technologies, Arlington, VA, USA, 26–28 September 2021; pp. 57–70.
30. Lewenberg, Y.; Sompolinsky, Y.; Zohar, A. Inclusive Block Chain Protocols. In *Proceedings of the Financial Cryptography and Data Security, San Juan, Puerto Rico, 26–30 January 2015*; Böhme, R., Okamoto, T., Eds.; Springer: New York, NY, USA, 2015; pp. 528–547.
31. SPECTRE: A Fast and Scalable Cryptocurrency Protocol. 2016. Available online: <https://eprint.iacr.org/2016/1159> (accessed on 4 October 2026).
32. Keidar, I.; Kokoris-Kogias, E.; Naor, O.; Spiegelman, A. All You Need is DAG. In Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing, Online, 26–30 July 2021; pp. 165–175.
33. Spiegelman, A.; Giridharan, N.; Sonnino, A.; Kokoris-Kogias, L. Bullshark: DAG BFT Protocols Made Practical. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022; pp. 2705–2718.
34. Danezis, G.; Kokoris-Kogias, L.; Sonnino, A.; Spiegelman, A. Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus. In Proceedings of the Seventeenth European Conference on Computer Systems, Rennes, France, 5–8 April 2022; pp. 34–50.
35. Ethereum Proof-of-Stake Attack and Defense. 2026. Available online: <https://ethereum.org/bg/developers/docs/consensus-mechanisms/pos/attack-and-defense/> (accessed on 4 October 2026).
36. Timeline of All Ethereum Forks. 2026. Available online: <https://ethereum.org/ethereum-forks/> (accessed on 4 October 2026).
37. Attestation. 2026. Available online: <https://ethereum.org/bg/developers/docs/consensus-mechanisms/pos/> (accessed on 4 October 2026).
38. Castro, M.; Liskov, B. Practical Byzantine Fault Tolerance. In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, New Orleans, LA, USA, 22–25 February 1999; pp. 173–186.
39. Brewer, E.A. Towards robust distributed systems. In Proceedings of the 19 Annual ACM Symposium on Principles of Distributed Computing, Portland, OR, USA, 16–19 July 2000; pp. 1–7.
40. Zheng, P.; Xu, Q.; Zheng, Z.; Zhou, Z.; Yan, Y.; Zhang, H. Meepo: Sharded Consortium Blockchain. In Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 19–22 April 2021.
41. Hong, Z.; Guo, S.; Zhou, E.; Zhang, J.; Chen, W.; Liang, J.; Zhang, J.; Zomaya, A. Prophet: Conflict-Free Sharding Blockchain via Byzantine-Tolerant Deterministic Ordering. In Proceedings of the IEEE INFOCOM 2023–IEEE Conference on Computer Communications, New York City, NY, USA, 17–21 May 2023.
42. Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y.; et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 23–26 April 2018.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.