Dolphin: Efficient Non-Blocking Consensus via Concurrent Block Generation

Xuyang Liu, Kaiyu Feng, Zijian Zhang, *Senior Member, IEEE*, Meng Li, *Senior Member, IEEE*, Xi Chen, Wenqian Lai, and Liehuang Zhu, *Senior Member, IEEE*

***Abstract*—Blockchain technology has become a research hotspot in distributed systems, aiming to sustain a decentralized ledger via consensus. Traditional consensus solutions exhibit slow processing speed and response time, resulting in poor performance. To address this issue, several consensus protocols have been proposed. One such popular protocol is HotStuff, a Byzantine fault-tolerant consensus (BFT) that achieves high throughput at the cost of latency. However, its throughput suffers from a proportional decrease with the increase in latency, posing a significant challenge. In this paper, we propose a new protocol called Dolphin that builds upon HotStuff. It operates in a partially synchronous network with n replicas, up to f byzantine faults, where $n \geq 3f + 1$, and achieves higher throughput in high-latency environments by leveraging non-blocking concurrent block generation. Specifically, we formalize our strategy as a generic Asynchronization Procedure Patch and prove that it does not affect the execution process of the original protocol. Theoretical analysis validates that Dolphin preserves the safety, liveness, and responsiveness properties while enhancing the throughput. The evaluation demonstrates that Dolphin typically achieves more than 10x higher throughput in Wide Area Network (WAN) environments with lower latency compared to HotStuff and its variants, and exhibits similar bandwidth utilization to DAG-based protocols such as Narwhal.**

***Index Terms*—Blockchain, Byzantine fault-tolerant, consensus, high-latency environments, non-blocking.**

Manuscript received 7 February 2024; revised 8 May 2024; accepted 8 May 2024. Date of publication 13 May 2024; date of current version 5 November 2024. The work of Xuyang Liu was supported by China Scholarship Council (CSC) under Grant 202306030197. This work was supported in part by the Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing , and in part by the National Natural Science Foundation of China (NSFC) under Grant 62172040, Grant 62372149, and Grant U23A20303. Recommended for acceptance by A. Garcia-Saavedra. *(Corresponding authors: Zijian Zhang; Meng Li.)*

Xuyang Liu is with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Haidian District 100081, China, and also with the School of Computer Science, The University of Auckland, Auckland 1010, New Zealand (e-mail: liuxuyang@bit.edu.cn).

Kaiyu Feng, Xi Chen, Wenqian Lai, and Liehuang Zhu are with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Haidian District 100081, China (e-mail: fengkaiyu@bit.edu.cn; chenxibit@bit.edu.cn; 1120212680@bit.edu.cn; liehuangz@bit.edu.cn).

Zijian Zhang is with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing, Haidian District 100081, China (e-mail: zhangzijian@bit.edu.cn).

Meng Li is with the Key Laboratory of Knowledge Engineering with Big Data (Hefei University of Technology), Ministry of Education, Hefei 230002, China, also with the School of Computer Science and Information Engineering, Hefei University of Technology, Hefei 230002, China, and also with the Anhui Province Key Laboratory of Industry Safety and Emergency Technology, Intelligent Interconnected Systems Laboratory of Anhui Province (Hefei University of Technology), Hefei 230002, China (e-mail: mengli@hfut.edu.cn).

This article has supplementary downloadable material available at https://doi.org/10.1109/TMC.2024.3399772, provided by the authors.

Digital Object Identifier 10.1109/TMC.2024.3399772

I. Introduction

In 2008, Nakamoto introduced the concept of blockchains [1] in his white paper on Bitcoin. This proposed a distributed ledger that could be maintained without a centralised authority. Since then, blockchain technology [2], [3] has become a prominent field within distributed systems research. Under the assumption that certain nodes may be malicious or down, this distributed ledger requires a specific consensus algorithm to keep consistency through the entire networks. However, traditional consensus protocols have been criticized for poor performance and high energy consumption [4], [5].

To address these issues, several types of protocols have been proposed, including PoS [6], [7], DPoS [8] of synchronous networks. Under partially synchronisation [9], PBFT [10] became the first practical consensus protocol that could achieve high throughput, low latency, and deterministic finality. However, PBFT has drawbacks such as high communication complexity. SBFT [11] uses threshold signatures [12], [13] to reduce communication complexity, Tendermint [14] locks the block to get a simpler view-change protocol, and Casper [15] as well as many other protocols using the idea of pipelining to further reduce the average communication cost of blocks.

HotStuff [16] combines the ideas of all the aforementioned protocols and introduces the property of responsiveness to propose a basic structural framework for simplifying the BFT protocol. It trades for high throughput with a small amount of latency using a three-phases protocol. Many variant protocols have been proposed to optimize the performance of HotStuff [17], [18], [19], [20], [21], [22], [23], [24]. An effective approach is to achieve better performance using fewer rounds. Marlins [24] and Jolteon [20] attempt to use two rounds to achieve finality while maintaining linearity in leader rotation (view-change). The normal technique is to separate the consensus into a happy path and an unhappy path, with the former being a fast path that could achieve finality in two rounds, and the latter being traded off but rarely triggered. Fast-HotStuff [22] and Narwhal [25] have also focused on the necessity of aggregated signatures [26] and possible alternative strategies, as this method consumes a large amount of computing resources, which can no longer be ignored when throughput is high. WPBFT [19] use random selection of leaders to reduce network communication complexity in HotStuff. A poster [23] in CCS' 22 propose a mitigation based on coding to let the leader encode the block into small chunks thus improving efficiency.

However, existing variants [17], [18], [19], [20], [21], [22], [23], [24] still cannot solve the main problem of throughput decreasing proportionally with increasing latency in HotStuff. This means that in environments with high latency (such as wide area networks, WANs [27], [28]), throughput is much lower than expected. DAG approach, primarily brought by Aleph [29], is a promising approach to solve this problem. DAG-Rider [30], Narwhal [25] and Bullshark [31] achieve zero communication overhead and high throughput while maintaining scalability and low latency. Since this series of solutions needs to be based on abstracting the network communication layer from consensus, this raises the question of how to achieve better performance similar to DAG solutions without changing consensus logic.

While BFT protocols, such as HotStuff, are elegant and widely used in production, they are not optimized for high-latency environments. In contrast, DAG-based protocols [25], [29], [30], [31] are purpose-built to overcome the obstacles presented by high-latency environments. However, the reconstruction of their network communication layer [30] requires additional efforts (i.e., handling a two-layer structure, where the first layer constructs a communication DAG through reliable broadcast, and the second layer observes and sorts all proposals based on this DAG. These two layers need to be handled independently), which introduces more complexity to the system. As a result, striking a balance between minimal interference with existing BFT systems and enhancing their performance in high-latency environments is a challenge that we aim to address. We also hope to formalize an independent strategy from the improved solution, which can maximize network bandwidth utilization while maintaining the integrity of the BFT protocol's state machine. thereby modularizing the protocol architecture.

In this paper, we propose a new BFT protocol called Dolphin. It operates in a partially synchronous network with n replicas, up to f byzantine faults, where $n \geq 3f+1$. It is a non-blocking protocol and achieves high throughput in high-latency environments. Dolphin maintains the safety, liveness, and responsiveness properties as the original HotStuff protocol, but with a new approach to improve blockchain performance in system throughput and consensus latency. Instead of requiring a certain number of votes from the previous view (round), the leader can propose extra blocks called in-between blocks that does not affect the state machines of replicas while waiting for consensus on the regular block proposed. The specific contributions are summarized as follows.

- We propose Dolphin, a BFT protocol to achieve high throughput in high-latency environments while preserving the properties of the original HotStuff protocol. The innovation of Dolphin lies in allowing the leader to propose in-between blocks that do not affect the state machines of replicas, which improves the performance of HotStuff and its variants.
- We formalize our strategy in Dolphin as a generic Asynchronization Procedure Patch for partially synchronous BFT protocols. We theoretically prove that this patch does not affect the execution process of the original protocol. On this basis, we proved that the Dolphin protocol preserves the safety, liveness and responsiveness properties as the original protocol.
- To demonstrate the effectiveness of our approach, we implement and evaluate Dolphin in a WAN environment with up to 100 nodes. Our results show that with minimal modifications, Dolphin achieves more than 10x higher throughput with lower latency compared to HotStuff and its variants, and can maximize network bandwidth utilization, similar to DAG-based protocols.

In a nutshell, Dolphin operates as follows: the protocol proceeds in views, with a dedicated leader per view. The leader proposes key blocks containing the latest state and a quorum certificate aggregated from the previous view. Replicas vote on these key blocks to reach consensus on the new state. However, the leader can also propose in-between blocks containing additional commands (transactions or client requests) between consecutive key blocks. These in-between blocks do not require votes and are immediately appended to the chain, allowing the protocol to make progress without waiting for consensus on the key block. The validity of them inherently tied to that of the key block. In-between blocks are processed in tandem with key blocks, a replica will cast votes to confirm the key block in the two views following the proposal of the key block if and only if both the in-between blocks and the key block proposed by the same leader in the proposal's view are deemed valid. After two confirmations, both types of blocks can be committed and their transactions can be finalized.

II. Related Work

Concurrency, Pipelining, and Non-blocking Vote: In consensus protocols where leadership is not rotated or fixed within a specified period, a technique known as concurrent consensus may be adopted. This technique allows for concurrent execution of multiple consensus instances, which can significantly increase the speed of the protocol. Protocols such as PBFT [10] and SBFT [11] can utilize this method since they make separate decisions on transactions. However, this approach has some drawbacks, such as its inability to be used when the leader is changing, and some votes maybe redundant due to the relationship between each block.

Casper [15], as well as HotStuff [16], utilize pipelining to address this issue. These protocols were designed with the inherent properties of blockchains in mind, allowing the leader to continuously generate new blocks while moving the previous block to the next stage, thus increasing throughput. However, it has been observed that this approach may result in a trade-off between bandwidth and latency, as consensus must still be reached through the propose-vote pattern.

Non-blocking Operations [32], [33], [34] is another term used in protocol design, as seen in Sync-HotStuff [18]. This refers to a set of operations that do not impact the critical path of the protocol, thus not affecting its latency. In general, the inclusion of non-blocking operations is a design decision that balances performance and implementation complexity.

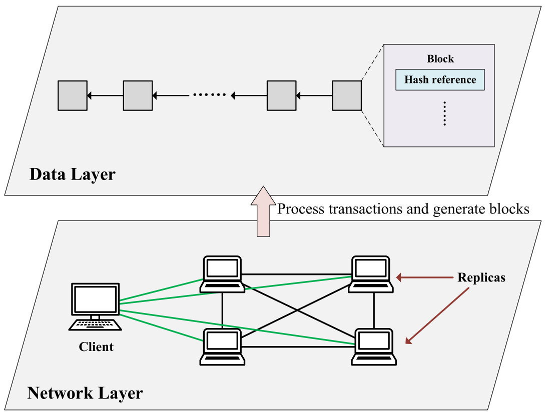


Fig. 1. System model.

Rather than utilizing non-blocking votes like in other protocols, our approach utilizes non-blocking proposals to improve throughput. Unlike letting multiple consensus instances run concurrently, our method eliminates the redundant votes by only duplicating propose operations.

Finality Gadget: Casper [15] and GRANDPA [35] proposed the concept of finality gadget. The finality gadget is a part of the consensus engine that only deals with finality of blocks. Since the consensus protocol is divided into two parts, block generation and block finality, it can be said that block finality is non-blocking with respect to block generation. However to our knowledge, finality gadget is brought up mainly to solve the problem of modularity. It focuses on reducing the complexity when a blockchain system wants to adopt multiple consensus protocols or make transition from one to another. In our work, we try to decouple the block generation and block finality in order to improve throughput.

Skipping Blocks: While Casper [15] also proposes the idea of skipping blocks to allow replicas to coordinate on voting for a future block, Dolphin takes a different approach. Instead of skipping a fixed number of blocks to allow replicas to coordinate on the next block to vote on, Dolphin skips a variable number of blocks by dynamically generating a variable number of in-between blocks based on current network conditions and client load. These in-between blocks do not require voting from replicas, allowing Dolphin to achieve higher throughput, especially in high-latency environments. Furthermore, Dolphin maintains the linear chain structure of traditional blockchain protocols, avoiding the complexity introduced by the GHOST rule and fork choice in Casper.

Block Categorization: Block categorization is a prevalent idea in the blockchain industry, as exemplified by Bitcoin-NG's Key Blocks and Microblocks [36] (similar concepts can also be found in Conflux [37], and Ethereum [38] 2.0 (Serenity) etc.). Unlike Dolphin, where consensus is achieved through a voting mechanism, Key Blocks in Bitcoin-NG are produced via a Proof of Work (PoW) method. Additionally, once a leader is elected through a Key Block in Bitcoin-NG, they are capable of rapidly generating Microblocks to enhance the system's throughput, a similarity shared with Dolphin. However, a distinctive feature of Dolphin is the parallel generation of in-between blocks which occurs without affecting the state machine replication. This facilitates a non-blocking block production, thereby improving system efficiency.

Shared Mempool: The technique of Shared mempool [25], [31], [39], [40], which originated in engineering for decoupling and was subsequently introduced into academic research for improving throughput under wide area networks (WANs) through concurrent transfer of transaction content while consensus is established, offers the additional benefit of scalability. However, this decoupling also introduces the potential for transactions to propagate outside of the constraints of consensus, thus creating the possibility of new attacks. In contrast, our method involves minimal changes to the original algorithm.

III. Problem Statement

A. System Model

A BFT system typically comprises two layers: the network layer and the data layer. The network layer consists of n replicas connected to each other to form a consensus group, a fraction of which may fail arbitrarily (Byzantine failures [41]). All replicas process transactions submitted by clients and generate blocks while ensure the global consistency of the operation logs for the data layer. Once waiting a period of time or reaching the max batch size (maximum batch size and waiting period are configurable parameters set based on the desired block size and latency targets), the transactions are packed in the form of a block b and being handled by replicas. Each replica has a public/private key pair and uses the public key as identity. The group executes a multi-phase consensus protocol to generate and broadcast blocks. In a period of time, generally called view, there is a replica as the leader to drive the protocol one round or more. We use an integer number $viewNumber$ to designate the replica ($viewNumber$ mod n) in a round-robin order. In the data layer, which is composed of the blocks stored by replicas, each block contains a hash reference to the previous block, forming a linear chain. Replicas maintain the chain of blocks by appending new blocks to the end of the chain. We show the system model in Fig. 1.

B. Threat Model

We consider the system that features a maximum of f replicas under the control of an adversary or down, with the number of compromised replicas not exceeding one-third of the total number of replicas. For simplicity, we let $n = 3f + 1$ be the total number of replicas, and at least $2f + 1$ nodes are honest. The compromised replicas exhibit Byzantine faults and may behave arbitrarily, such as violating the protocol or refusing to respond. They can eavesdrop on network communication but lack the ability to impede or alter messages during point-to-point transmission. The remaining non-faulty replicas communicate honestly with each other and execute the protocol correctly. We also simplify the protocol by requiring that every message is signed by the sender's private key and verified by the corresponding public key by the receiver.

C. System Goals

Our system employs a protocol that consistently decides on transactions from users in chronological order. To ensure the protocol's effectiveness, the protocol needs to satisfy the following properties:

- *Safety:* All non-faulty replicas have the same view of the system and the same ordered log of transactions.
- *Liveness:* A correct client's request is eventually delivered and committed by all non-faulty replicas.
- *Responsiveness:* The protocol can generate the honest leader to push consensus within the actual network delay, rather than waiting for the maximum value.

Where safety is guaranteed even in asynchronous networks, but liveness relies on synchronous network assumptions. Responsiveness means the leader does not need to wait a long time for potential conflicts to take steps.

D. Network Assumptions

Our protocol is designed to be run in a partially synchronous network [9], which is bounded by a known finite time bound Δ and a special time event known as the global stabilization time (GST). After GST, all transmissions between two correct replicas arrive within time Δ, and the adversary must cause the GST event to eventually happen after some unknown finite time. It should be noted that the Asynchronization Procedure Patch method discussed in Section VI is only applicable to partially synchronous networks. It is a method that improves the protocol by introducing asynchronous block proposals. This leads to the distinction between synchronous and asynchronous protocols.

a) Synchronous protocol: In a synchronous protocol [10], [14], [16], a valid proposer must wait for vote messages from other validators before proposing a block (with transactions), resulting in a time delay of at least δ_n between consecutive proposals, which is the time lower bound of the network.

b) Asynchronous protocol: In contrast, in an asynchronous protocol [29], a valid proposer can propose a new block at any time upon receiving client commands. In the best case scenario, the time delay between consecutive proposals is δ_c, which is the computation time of a node. Typically, δ_c is much smaller than δ_n.

IV. Preliminaries

In this section, we briefly introduce a BFT system named Hotstuff [16], showing the data structures, phases, and utilities.

A. Data Structures

Quorum certificates: A quorum certificate (QC) over a tuple $< type, viewNumber, node >$ is a set of signed votes on a block from a quorum of replicas ($2f+1$ in our case), which can be implemented using aggregate signature. In our work, we adopt aggregate signatures as in Hotstuff in our pseudocode to demonstrate the key differences between our method and HotStuff, but in the implementation, we choose to avoid this costly operation. We define the following syntax for QC:

- The symbol "$\Leftarrow$" denotes the node a QC refers to, where $b.justify.node \Leftarrow b$.

Blocks: We assume that the consensus protocol is utilized in the blockchain system. A block is considered valid if its parent block is valid and it satisfies both the application-specific rules and the consensus rules. In order to elaborate on the relationship between blocks, we define the following terms:

- $b_i \leftarrow b_j$ if b_i is the parent of b_j.
- $b_i \overset{\bullet}{\leftarrow} b_j$ if b_i is the ancestor of b_j and the QC of b_j is that of b_i. In other words, there is no other key blocks between them.
- $b_i \overset{*}{\leftarrow} b_j$ if b_i is the ancestor of b_j

Tree and branches: A tree T is a hierarchical data structure where each node has exactly one parent, except for the root node which has no parent. A branch is a sequence of connected nodes that starts from the root node and ends at a leaf node in the tree. Two branches are conflicting if neither one is an extension of the other. Two nodes are conflicting if the branches led by them are conflicting. Typically in blockchain systems, nodes are blocks.

Messages: A message m in the protocol includes fixed fields populated using the Msg utility. Each message has a naming type ($m.type$) determined by the phase and a proposed node ($m.node$). $m.justify$ is an optional field used by the leader to carry the QC and by replicas in NEW-VIEW messages to carry the highest $prepareQC$. Messages sent by a replica include a partial signature ($m.partialSig$) by the sender over the tuple $< m.type, m.viewNumber, m.node >$, added in the VOTEMSG utility.

B. Phases

The Hotstuff protocol uses four phases to achieve consensus: PREPARE, PRE-COMMIT, COMMIT, and DECIDE. A new leader starts by collecting NEW-VIEW messages from (n-f) replicas. The NEW-VIEW message is sent by a replica as it transitions into $viewNumber$. The NEW-VIEW message includes a QC referencing the highest QC ($\perp$ if none) seen so far. In chained hotstuff, the four phases are unified into the GENERIC phase of the same pattern.

GENERIC phase: The GENERIC phase refers to a phase that can be used to perform any operation that is not covered by the prepare, pre-commit, commit, or decide phases. During the generic phase, the leader collects $n-f$ generic messages from replicas in the previous phase, and then aggregates them into QC for broadcasting in the current phase. Replicas can perform any necessary operations to maintain the integrity and consistency of the system. The generic phase is an important component of the Hotstuff protocol, allowing for greater flexibility and adaptability in a variety of distributed systems.

NEXTVIEW interrupt: A replica waits for a message at view $viewNumber$ for a timeout period in generic phase, determined by an auxiliary NEXTVIEW utility. If NEXTVIEW interrupts waiting, the replica will also increment $viewNumber$ and start the next view.

C. Utility Functions

MSG $(type, node, qc)$. This function creates a message with given $type$, proposed block $node$, qc and current $viewNumber$. It returns the message.

VOTEMSG $(type, node, qc)$. For the given parameters, this function calls MSG to generate message m and generates a partial signature of the caller stored in $m.partialSig$.

CREATELEAF $(parent, cmd, qc)$. This function creates a new block with a given parent block, a command from client and a qc. It returns the new block.

QC (V). This function creates a qc from a set of messages V. The qc includes a message type $type$, a view number $viewNumber$, and proposed block nodes $node$ from each message in V, as well as a combined signature from all the messages.

MATCHINGMSG (m, t, v). This function checks if a message m has a given type t and a view number v. It returns true if the message matches the given type and the view number, and false otherwise.

MATCHINGQC (qc, t, v). Similar to MATCHINGMSG, one of the check parameters is replaced with qc from message m.

SAFENODE $(node, qc)$. This function checks if a proposed node satisfies the safety or liveness rules. It returns true if the node extends from the $lockedQC.node$, or if the $viewNumber$ in qc is greater than $lockedQC.viewNumber$.

V. Dolphin Protocol

In order to comprehensively introduce the Dolphin protocol and asynchronize block generation, we will take some additional steps to asynchronize the synchronous block generation process in a more intuitive manner.

In Section V-A, we asynchronize the synchronous protocol from the leader's perspective. Then, in Section V-B, we discuss how a replica should deal with these asynchronous messages and introduce the concepts of key blocks and in-between blocks in Section V-C. Using these concepts, we provide a detailed description of the Dolphin protocol in Section V-D, including the data structure that needs to be modified compared to the HotStuff and the specific process of the Dolphin. We also provide pseudocode to implement Dolphin and analyze the protocol's efficiency.

A. Concurrent Block Generation

Ideally, the throughput of the protocol is determined by the block generation rate. We noticed that the block generation is currently coupled with the voting process, which can be a bottleneck for throughput. In recent years, several protocols have attempted to address this issue by applying pipelining to utilize multiple states before block finalization, such as Casper and HotStuff. However, these approaches still require at least one round of voting to block the next proposal.

An alternative approach is to introduce concurrency to the consensus process by allowing multiple consensus processes to occur simultaneously. However, this approach raises several issues that must be addressed in order to maintain safety and liveness properties, e.g. block ordering issues, possible fork handling, etc. Another path is to concurrent a consensus, by allowing multiple consensus precedes at the same time. But it'll lead to several main problems, if not solved properly. First, replicas would need to vote multiple times in a given view, which could lead to conflicts with the protocol's safety properties. Second, implementing a protocol that achieves concurrency by copying every aspect of the original protocol would be extremely challenging. Finally, even if the protocol is altered to provide safety and liveness, this approach would lose its generality and would not be applicable to other protocols.

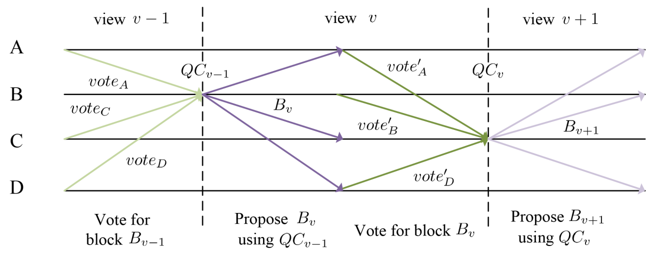


Fig. 2. The basic segment of HotStuff: the leader proposes a block (marked in purple) and replicas vote (marked in green) on it. Votes are sent to the next leader (supposing that two consecutive views have different leaders).

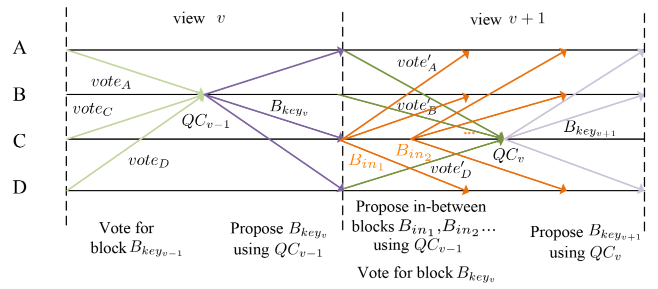

view v
view $v+1$
A
B
C
D
$vote_A$
$vote_C$
$vote_D$
QC_{v-1}
B_{key_v}
$vote'_A$
$vote'_B$
$vote'_D$
B_{in_1}
B_{in_2}
...
QC_v
$B_{key_{v+1}}$
Vote for block $B_{key_{v-1}}$
Propose B_{key_v} using QC_{v-1}
Propose in-between blocks B_{in_1},B_{in_2}... using QC_{v-1}
Vote for block B_{key_v}
Propose $B_{key_{v+1}}$ using QC_v

Fig. 3. The leader of view $v+1$ starts to propose new blocks immediately (marked in orange) when receiving a proposal from the last view, entering a new view and becomes the leader. This process does not block the original path and can be repeated until the leader collects enough votes to propose a key block. (the process of proposing in-between blocks should also be present in view v. We just omit it to emphasise the behaviour in view $v+1$).

We have simplified the state-of-the-art BFT protocol into Fig. 2. In each view, the leader proposes a block and sends it to all replicas, who then vote on it and send the votes back to the leader. For simplicity, we assume that the leader changes in a round-robin fashion. Whenever a replica receives a proposal or a vote, it processes it according to the protocol and updates its state, and attempts to move one or more blocks to the next stage. For safety and liveness, a block cannot be confirmed until several QC formed by sufficient votes from different views are received. But in the lens of message passing, these constraints are not shown in the figure for simplicity.

In this protocol, we attempt to decouple the generation and finalization of blocks. When the replica receives a proposal from the last view and becomes the leader in the new view, it can immediately generate new blocks. This is illustrated in Fig. 3, where the leader can generate a new block without waiting for responses from other replicas. The asynchronously non-blocking proposed messages are marked in orange. These

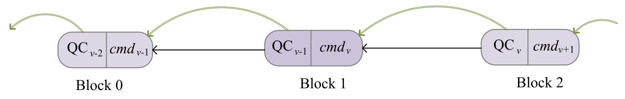

QC $_{v\text{-}2}$ | $cmd_{v\text{-}1}$
Block 0
QC $_{v\text{-}1}$ | cmd_v
Block 1
QC $_v$ | cmd_{v+1}
Block 2

Fig. 4. The typical blockchain structure is as follows: when there is no timeout, a block points to its parent block and contains a QC pointing to the preceding block (shown in green).

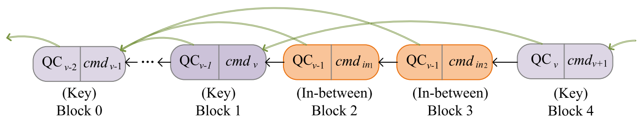

QC $_{v\text{-}2}$ | $cmd_{v\text{-}1}$
(Key)
Block 0
QC $_{v\text{-}1}$ | cmd_v
(Key)
Block 1
QC $_{v\text{-}1}$ | cmd_{in1}
(In-between)
Block 2
QC $_{v\text{-}1}$ | cmd_{in2}
(In-between)
Block 3
QC $_v$ | cmd_{v+1}
(Key)
Block 4

Fig. 5. In-between blocks, which are async-produced blocks, always have the same QC as the preceding block. The key block may carry a different QC, which will potentially promote the state machine to the next stage.

proposals propose blocks based on each other in a non-blocking manner, so in between the original proposals, several blocks are generated and proposed using an asynchronous way.

To provide a better understanding of the protocol, we can think of this step as a concurrent protocol, but with several differences. Firstly, the leader will mark these blocks with a new type to distinguish them from blocks in the critical path. Secondly, these blocks will carry the same QC as the previous one. Finally, instead of using a time window to determine leader rotation, a leader will stop proposing these in-between blocks when it gathers enough votes and can then propose the block in the critical path. Later, we will provide a strict definition of these two types of blocks and the Dolphin protocol.

B. Block Categorization and Voting Optimization

We identified critical problems in the native concurrent approach, which are primarily caused by abuse of the voting process. To illustrate the problem more clearly, we have included a diagram in Fig. 5 showing how the blocks are organized in the protocol. From the perspective of leader replica c in view v, after receiving block 1 and before gathering enough votes to form a QC for block 1, the blocks it generates (marked in orange) must carry the same QC as block 1. Thus for all replicas, upon receiving blocks 2 and 3, there is no new information to facilitate state transitions, except for adding them to the block tree (assuming all the blocks are valid in the Fig. 5).

We then distinguish the blocks into two categories **key blocks** and **in-between blocks**. Key blocks are the blocks that contain new QC gathered from other replicas by a leader. In-between blocks carry different commands (transactions or client requests) than key blocks, but it always carry a duplicate of the QC in the previous block. In Fig. 5, block 2.3 are in-between blocks. While the others are key blocks. As a comparison, we also provide a typical blockchain structure in Fig. 4, where every block always contains a new QC and can therefore all be considered as key blocks.

Furthermore, in the context of blockchains, voting is not required for every block in order to finalize it, since each block references the previous block, which implies the transmission of vote results.

By only voting on the key blocks when they are being processed, replicas can treat the in-between blocks as placeholder data for the key blocks, optimizing throughput without significantly increasing the complexity of the protocol.

C. Key Blocks and In-Between Blocks

a) Key Block: A key block is a block that is signed (or will be signed) by a quorum of replicas. It carries the new QC, so as shown in Fig. 5, the lowest block that contains a certain QC among several blocks is always a key block.

b) In-between Block: An in-between block is a block that is not a key block. Although it contains a QC, the QC is always a duplicate of the QC in the previous block. Informally, these blocks can be understood as additional chunks of information that are attached to the key block, and can be generated in a non-blocking manner.

From the perspective of voting, in-between blocks are the blocks that will not receive any votes in any case. The validity of them inherently tied to that of the key block. Upon receiving an in-between block, each replica evaluates its validity using the same criteria applied to key blocks, due to their structural similarity. A replica will continue to cast votes to confirm the key block in the views following the proposal of the key block if and only if both the in-between blocks and the key block proposed by the same leader in the proposal's view are deemed valid.

In-between blocks also do not affect the state machine. By "state machine", we mean the entire state preserved by a replica. There are a few conditions worth explaining further:

- The state machine of the consensus layer doesn't take into account how transactions are executed , as the consensus treats all the transactions as payloads for blocks.
- Even if the application-level layer marks an in-between block as an invalid block, this can be treated as an invalid descendant key block.
- The state machine is not concerned with how the block tree grows, but only how the states change.

It is important to note that in-between blocks do not affect the state transitions of the state machine, but this does not imply that modifications to them cannot be recognized by the state machine. Due to the characteristics of blockchain, any tampering with legitimate in-between blocks will result in a change in the hash value of the latter block, which ultimately impacts the hash value of the key block, leading to rejection in subsequent consensus processes.

In this context, we define $\text{HEIGHT}(B_n^{key})$ as the number of ancestor key blocks of B_n^{key}. We say that two key blocks $B_i^{key} \approx B_j$, if all the following conditions are satisfied:

- $\text{HEIGHT}(B_i^{key}) = \text{HEIGHT}(B_j)$.
- The states of the two blocks are the same.
- B_i^{key} in asynchronous protocol and B_j in synchronous protocol.
- $B_i^{key}.justify.node \approx B_j.justify.node.$

Finally, we say two block trees $T \approx T'$ if the following condition is satisfied: For any B_i^{key} in T, there exists a unique and unrepeatable block B_j such that $B_i^{key} \approx B_j$.

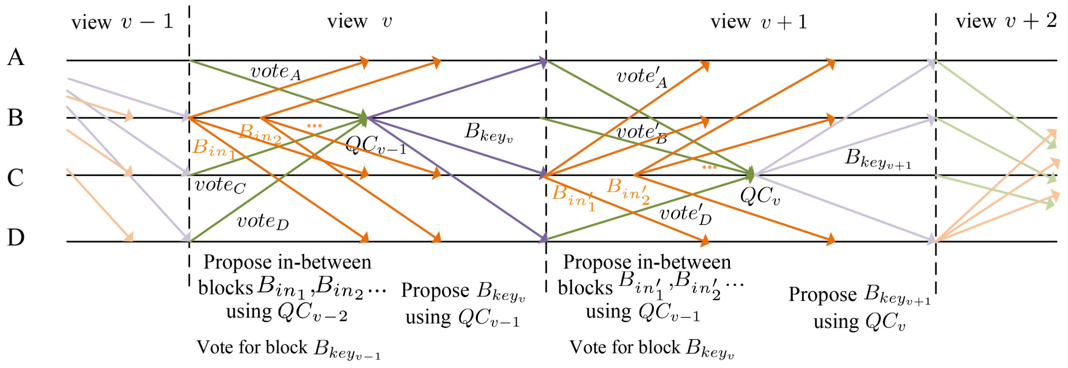

view $v-1$
view v
view $v+1$
view $v+2$
A
B
C
D
$vote_A$
B_{in_1}
B_{in_2}
...
QC_{v-1}
B_{key_v}
$vote_C$
$vote_D$
$vote'_A$
$vote'_B$
$B_{in'_1}$
$B_{in'_2}$
...
$vote'_D$
QC_v
$B_{key_{v+1}}$
Propose in-between blocks B_{in_1},B_{in_2}... using QC_{v-2}
Propose B_{key_v} using QC_{v-1}
Vote for block $B_{key_{v-1}}$
Propose in-between blocks $B_{in'_1}$,$B_{in'_2}$... using QC_{v-1}
Propose $B_{key_{v+1}}$ using QC_v
Vote for block B_{key_v}

Fig. 6. Visualization of Dolphin: prior to accumulating sufficient votes on its view, leader will generate new in-between blocks and propose them to other replicas. Upon gathering enough votes, leader will propose the key block.

D. *Dolphin Protocol Specification*

A visualization of Dolphin is provided in Fig. 6. Before giving a formal definition of it, we will compare and contrast the modified data structures in the protocol with those in Chained HotStuff. The goal of Dolphin is to provide an asynchronous block-generation process while maintaining the other properties of HotStuff. To achieve this, we have made minimal modifications to the protocol. We have chosen Chained HotStuff as the base protocol because it is straightforward to illustrate how our protocol works. In this section, we will focus on how Dolphin achieve asynchronous block generation, and we will not discuss the protocol invariants, which are unchanged from the base protocol. Algorithm 1 presents the pseudocode for Dolphin.

1) *Modified Data Structures:*

a) Messages: Dolphin further divides the GENERIC message type into GENERIC$_{key}$ (Line 6) and GENERIC$_{in-between}$ (Line 21) messages in order to differentiate between key blocks and in-between blocks. Instead of waiting for $n-f$ votes from the previous view, the leader can generate new blocks immediately upon entering the new view and propose them to other replicas (Line 19–21). As long as the leader does not receive sufficient QC or reach a timeout, the process will continue (Line 22). These blocks generated by the leader are referred to as in-between blocks and are marked with GENERIC$_{in-between}$. After gathering a QC, the leader will transmit a GENERIC$_{key}$ message to the other replicas. This message serves the same purpose as the proposal message in the original protocol (Line 3–6).

b) Tracking the Parent Key Block: Due to the potential uncertainty in the number of in-between blocks between two key blocks, it is not possible for a replica to track whether key blocks referenced by consecutive QC receive votes in a consecutive view using a one-time hash. However, in Dolphin, the leader can differentiate between these blocks by message type and subsequently, track them.

Based on the definition of in-between blocks, we should only consider the relationship between key blocks. Therefore, the parent-child relationship between two key blocks, B_i^{key} and B_j^{key}, is equivalent to the relationship in the original protocol if $B_i^{key} \overset{\bullet}{\leftarrow} B_j^{key}$.

c) One-Chain, Two-Chain, and Three-Chain: When a key block b_i^{key} carries a QC that refer to another key block b_j^{key} such that $b_j^{key} \overset{\bullet}{\leftarrow} b_i^{key} \wedge b_i^{key}.justify.node = b_j^{key}$, we say that it forms a *One-Chain* (Line 12). When a key block b_k^{key} and ${}_j^{key}$ forms another *One-Chain*, $b_i^{key}, b_j^{key}, b_k^{key}$ form a *Two-Chain*. Similarly, another *One-Chain* in addition to the previous two *Two-Chains* forms a *Three-Chain*. Note that in-between blocks are not considered since they would never receive votes or be processed by the consensus algorithm (Line 8, Line 11). Under certain circumstances, a replica may receive a QC from block b_i^{key} that refers to a key block $b_{i'}^{key}$ that does not satisfy the condition $b_{i'}^{key} \overset{\bullet}{\leftarrow} b_i^{key}$. This may occur in the case of a network partition, replica crash, invalid block, or malicious behavior.

For blocks $b_i^{key}, b_j^{key}, b_k^{key}, b_x^{key}$ that have a relationship $b_x^{key} \Leftarrow b_k^{key} \Leftarrow b_j^{key} \Leftarrow b_i^{key}$, if a *One-Chain* forms from b_k^{key}, then b_x^{key} and all the previous blocks, including in-between blocks, are considered to have completed the PREPARE state. The $genericQC$ should be updated to $b_x^{key}.justify$ (Line 12-13). If a *Two-Chain* forms from b_j^{key}, then b_x^{key} and all the previous blocks, including in-between blocks, are considered to have completed the PRE-COMMIT state. The $lockQC$ should be updated to $b_x^{key}.justify$. (Line 14-15) Finally, if a *Three-Chain* forms from b_i^{key}, then b_x^{key} and all the previous blocks, including in-between blocks, are considered to have completed the COMMIT state. All commands in these blocks should be executed (Line 16-17).

2) *Dolphin Protocol:*

It is important to note that several assumptions are made for the sake of brevity:

- The QC should always refer to its ancestor block.
- An ancestor block should always be processed before its descendant block.
- All the blocks should be validated before being processed by the consensus algorithm.

If any of these conditions are not satisfied, the basic security property may be violated. In such a case, a replica should stop voting for the affected block and its descendant blocks. More strict rules, such as equivocation detection and slashing, may also be applied, but they are beyond the scope of this paper.

3) *Efficiency Analysis:*

We denote the network delay as δ_n and the computation delay as δ_c, and it is clear that $\delta_n \gg \delta_c$. By

Algorithm 1: Dolphin Protocol.

```
1:  for $curView \leftarrow 1, 2, 3 \cdots$ do
      ▷ GENERIC phase
2:    as a leader do ▷ $r$ = LEADER($curView$)
        ▷ $M$ is the set of messages collected at the end of previous view by the leader of this view
3:      $highQC \leftarrow \left( \arg\max_{m \in M} \{m.justify.viewNumber\} \right) .justify$
4:      if $highQC.viewNumber > genericQC.viewNumber$ then $genericQC \leftarrow highQC$
5:      $curProposal \leftarrow$ CREATELEAF($genericQC.node$, command, $genericQC$)
        ▷ PREPARE phase (key blocks)
6:      broadcast MSG(GENERIC$_{key}$, $curProposal$, $\perp$)

7:    as a replica do
8:      wait for message $m$: MATCHINGMSG($m$, GENERIC$_{key}$, $curView$) from LEADER(curView)
9:      $b^* \leftarrow m.node; b'' \leftarrow b^*.justify.node; b' \leftarrow b''.justify.node; b \leftarrow b'.justify.node$

10:     if SAFENODE($b^*$, $b^*.justify$) then
11:       send VOTEMSG(GENERIC$_{key}$, $b^*$, $\perp$) to LEADER($curView + 1$)
        ▷ start PRE-COMMIT phase on $b^*$'s parent key block
12:     if $b'' \overset{\bullet}{\leftarrow} b^*$ then
13:       $genericQC \leftarrow b^*.justify$
          ▷ start COMMIT phase on $b^*$'s grandparent key block
14:       if $b' \overset{\bullet}{\leftarrow} b''$ then
15:         $lockedQC \leftarrow b''.justify$
            ▷ start DECIDE phase on $b^*$'s great-grandparent key block
16:         if $b \overset{\bullet}{\leftarrow} b'$ then
17:           Execute new commands through $b$, respond to clients

18:   as the next leader do
19:     repeat
          ▷ PREPARE phase (in-between blocks)
20:       $curProposal \leftarrow$ CREATELEAF($genericQC.node$, command, $genericQC$)
21:       broadcast MSG(GENERIC$_{in-between}$, curProposal, $\perp$)
22:     until for all messages: $M \leftarrow \{m | $MATCHINGMSG($m$, GENERIC$_{key}$, $curView$)$\}$
          until there are $(n - f)$ votes: $V \leftarrow \{v \mid v.partialSig \neq \perp \wedge v \in M\}$
23:     $genericQC \leftarrow QC(V)$
      ▷ Finally
24:   NEXTVIEW interrupt: goto this line if NEXTVIEW($curView$) is called during "wait for" in any phase
25:   Send MSG(GENERIC, $\perp$, $genericQC$) to LEADER($curView + 1$)
```

improving the block proposing interval from $2\delta_n$ to δ_c, Dolphin is able to achieve higher throughput even when δ_n increases. The latency of Dolphin is similar to that of HotStuff. Since in-between blocks are finalized by their descendant key blocks and Dolphin follows the same critical path as Chained HotStuff, the end users will not notice a difference between the two protocols.

It is important to note that Dolphin still outperforms HotStuff when the leader changed every view, as shown in Fig. 11. In practice, we can achieve even much higher throughput by not changing leaders frequently.

4) Complexity: Dolphin and HotStuff have the same communication complexity per block. However, Dolphin increases the communication complexity per view by proposing additional in-between blocks. In each phase of Dolphin, the leader broadcasts to all replicas, while the replicas respond to the sender once with a partial signature to certify the vote for key blocks. The key difference lies in the generation of in-between blocks, where the leader can continuously propose new in-between blocks without collecting votes from other replicas. Therefore, for each view, the communication complexity for the key block remains $O(n)$, the same as HotStuff. However, the propose of in-between blocks between two consecutive key blocks introduces additional complexity for broadcasting these blocks. Nonetheless, this overhead is compensated by the improved throughput achieved by Dolphin, especially in high-latency environments. The overall complexity per view is $O((1 + k)n)$, where k is the number of in-between blocks generated within that view (but for each block, the complexity is still $O(n)$).

Another advantage of Dolphin is its ease of implementation by modifying the original protocol, as they share the same code for the critical path. However, this also means that Dolphin requires introduces additional space complexity compared to HotStuff to track in-between blocks before the next key block proposal. Specifically, if k is the number of in-between blocks generated between two key blocks, and s is the size of a single block, then the additional space complexity introduced by Dolphin is $O(k * s)$. However, this overhead is temporary and can be controlled by limiting the value of k.

In general, there is no need to limit the maximum number of in-between blocks between two key blocks. The leader can continue proposing in-between blocks until it collects enough votes to propose a key block. However, if memory or bandwidth is limited, an artificial upper bound on the number of in-between blocks between two key blocks can be set. From the memory resources perspective: the upper bound on k should be set such that the total space required to track k in-between blocks, along with the key blocks, does not exceed the available memory on the replicas. Let Mem be the available memory per replica when no in-between blocks are proposed, and s be the size of a single block. Then, k can be bounded as: $k \leq (Mem - s)/s$

From the communication bandwidth perspective: Since the leader needs to broadcast all in-between blocks to the replicas, the value of k should be limited to ensure that the communication overhead does not saturate the available network bandwidth. Let $Band$ be the available bandwidth when no in-between blocks are proposed, and δ_c be the computation delay (or the block proposing interval). Then, k can be bounded as: $s \times k/\delta_c \leq Band$, s is the size of a single in-between block. In practice, the upper bound on k should be set to the minimum of the two bounds derived above, accounting for both memory and bandwidth constraints.

VI. Proof of Correctness

In this section, We first formalize our strategy in Dolphin as a generic Asynchronization Procedure Patch to reflect Dolphin's modifications compared to HotStuff, and prove that this patch does not affect the execution process of the original protocol. On this basis, we subsequently proved that the Dolphin protocol preserves the safety, liveness and responsiveness properties as the original protocol.

Algorithm 2: Asynchronization Procedure Patch.

```
1: procedure ASYCHPROPATCH(PROTOCOL)
       ▷ Concurrent block generation
2:     Copy the block generation procedure of PROTOCOL as BLOCKSGEN.
3:     Change the process of collecting proofs for a new block in BLOCKSGEN to copying existing proofs from the previous block. Then change the generated block type to in-between block to get inbetweenBlocksGen
4:     Loop inbetweenBlocksGen until PROTOCOL's waiting stops.
       ▷ Block categorization and voting optimization
5:     Modify rest part of the PROTOCOL to ensure that the protocol does not have to collect votes for in-between blocks, and can distinguish between two types of blocks. We then get the patched protocol PROTOCOL'
6:     Return PROTOCOL'
```

A. Asynchronization Procedure Patch

Asynchronization Procedure Patch requires that the origin protocol fulfill the following assumptions:

- The origin protocol is a (partially) synchronous protocol. (Asynchronous protocols may not have improved throughput, but the same properties still apply).
- The origin protocol is a block-based protocol, where the state change of a block potentially affects the state of its ancestors.

In-between blocks are blocks that can be generated in a non-blocking manner and do not affect the consensus state of a replica. This means that the state of the replica remains unchanged, except for the growth of the block tree and the content of the blocks. The definition of in-between blocks also acquires that, if a protocol relies on the block tree structure (such as the number of blocks in the tree or the relationship between blocks), it should completely ignore in-between blocks when making comparisons.

The core strategy of Asynchronization Procedure Patch is shown in Algorithm 2 on Line 1. Asynchronization Procedure Patch modifies a protocol from the perspective of both block generation (by the leader) and blocks voting (by the voters). As shown in Lines 2 to 4, the leader will continuously generate in-between blocks until the critical path, as specified in the original protocol, can continue. Line 5 allows the voter to ignore in-between blocks completely, and only treat key blocks through the consensus algorithm.

Regarding the value of backward compatibility with existing partially synchronous schemes, when network delays increase, the time required for message transmission also increases, leading to a longer time interval between the proposal made by the leader and the receipt of the corresponding proposal's votes by the next leader. During this waiting period, most partially synchronous BFT protocols such as PBFT [10], HotStuff [16], Tendermint [14] remain idle, wasting network bandwidth. The Asynchronization Procedure Patch, on the other hand, takes full advantage of this idle time and can utilize the network bandwidth by continuously proposing in-between blocks, significantly improving throughput. the throughput T_s is limited by the latency of the network, as described by the equation $T_s = \frac{cB}{L}$, where B is the bandwidth, c is a constant, and L is the latency. Asynchronization Procedure Patch optimizes the throughput by calculating $T_a = c'B$. When the latency is negligible, the throughput of asynchronous and synchronous protocols is similar. However, when the latency is significant, the throughput of asynchronous protocols is not significantly affected compared to the synchronous version.

However, this approach does require a trade-off in terms of memory consumption in order to track in-between blocks, as there may be hundreds of them between two key blocks. This trade-off can be adjusted by limiting the number of in-between blocks. In the case of a leader change, the generation process of in-between blocks will be delayed until the key block is received by the new leader, as shown in Fig. 6. Otherwise, the next leader may potentially create a fork in the blockchain.

In practice, there are several ways to implement in-between blocks, e.g. using a unique proposal message type to hold in-between blocks, or using an extra field in the block's data structure to mark the difference, etc. No matter which way we choose, ultimately, the voter can then build a virtual block tree consisting only of key blocks in the data layer, in order to minimize protocol changes. We reuse the notations and their definitions in Section V, a block tree T is the block tree of the patched protocol, and T' is the block tree of the original protocol.

Lemma 1: In-between blocks will not affect the origin HotStuff protocol's state machine.

Proof: In line 19 of Algorithm 1, the leader broadcasts a GENERIC$_{in-between}$ message to mark the in-between blocks. In line 8, a replica only matches and begins processing the key blocks. Therefore, the in-between blocks will not affect the protocol's state machine. □

Lemma 2: Under the same conditions, including a given $T \approx T'$, the leader of Dolphin will produce a block $B_i^{key} \approx B'_j$ produced by the leader of HotStuff.

Proof: As previously stated in pseudocode, the only difference between the two protocols is the method to create in-between blocks. Since Lemma 1, the in-between blocks do not affect the protocol's state machine under the same conditions. Thus, Dolphin will produce the key block $B_i^{key} \approx B'_j$ in HotStuff. □

Lemma 3: Under the same conditions, for a given $T \approx T'$, eventually, the two protocol will move their block trees into the next states such that $T_{new} \approx T'_{new}$

Proof: In accordance with Lemma 2, under a given $T \approx T'$, the replica will receive $B_i^{key} \approx B'_j$.

Then, we will enumerate all possible cases for the next state of the block tree as follows:

- Dolphin updates a $genericQC$ when *One-Chain* is satisfied, while HotStuff updates a $genericQC$ when *One-Chain* is satisfied.
- Dolphin updates a $lockedQC$ when *Two-Chain* is satisfied, while HotStuff updates a $lockedQC$ when *Two-Chain* is satisfied.

- Dolphin commits commands when *Three-Chain* is satisfied, while HotStuff updates a $executedQC$ when *Three-Chain* is satisfied.
- Both HotStuff and Dolphin vote under the same SAFENODE.
- The remaining code of Dolphin is the same as HotStuff.□

Theorem 1: Asynchronization Procedure Patch does not affect the execution process of the original Hotstuff protocol.

Proof: In this proof, we use mathematical induction to demonstrate that under the same conditions, given n key blocks, the block trees T and T' for two protocols will always be similar, denoted by $T \approx T'$.

a) Base Case: The definition of a key block states that $B_{genesis}^{key} \approx B'_{genesis}$.

b) Induction Step: As stated in Lemma 3, under the same conditions, for a given $T \approx T'$, the block trees of the two protocols will eventually reach the next states such that $T_{new} \approx T'_{new}$.

As both the base case and the induction step have been shown to be true, by mathematical induction, the theorem holds for all n key blocks. This implies that from the perspective of key block, the behavior of the Dolphin protocol is equivalent to that of HotStuff, and thus Asynchronization Procedure Patch does not affect the execution process of the original Hotstuff protocol.□

Theorem 1 is important because it demonstrates that Dolphin's key-block inherits the properties of the block in HotStuff. Therefore, our next goal is to analyze the impact of the in-between block on consensus proporties.

B. Safety, Liveness, and Responsiveness

Lemma 4: For any valid in-between blocks b_1, b_2, we have $b_1.qc \neq b_2.qc$

Proof: To show a contradiction, suppose $b_1.qc = b_2.qc$. This means that both b_1 and b_2 are in-between blocks in the same view v. Based on the properties of blockchain and hash functions, these two conflicting in-between blocks will result in their subsequent in-between blocks (if any) calculating completely different hash values, ultimately affecting the hash value of the next key block, resulting in conflicting key blocks in the same view $v + 1$. This is impossible, because in the original HotStuff protocol, two conflicting blocks must not be under the same view. According to Theorem 1, Dolphin's key blocks inherits this property. □

Theorem 2: (Safety) Let b_i and b_j be two conflicting blocks, then they cannot be both committed, each by a correct replica.□

Proof: This theorem includes two situations: both b_i and b_j are key blocks, or both b_i and b_j are in-between blocks. The former case holds according to Theorem 1 (In the original HotStuff protocol, conflicting blocks cannot be both commited, each by a correct replica). We prove the latter case by contradiction.

Let qc_i denote the QC in block b_i, and qc_j denote the QC in block b_j. By Lemma 4, $qc_i \neq qc_j$. Since in-between blocks contain the same QC as the previous block, their different qc means their previous key blocks are different. We will now denote by b'_i in view i' the first key block after b_i and b'_j in view j' the first key block after b_j. Due to the conflict between b_i and b_j, and the fact that a block in the chain structure blockchain cannot refer to two different blocks simultaneously, b'_i and b'_j must be two conflicting key blocks, which lead to the former case. So, these two conflicting in-between blocks cannot be both committed, each by a correct replica.

Lemma 5: After GST, there exists a time period P that all honest replicas remain in the same view v during P and the leader for view v is correct. An honest leader in such a view can always broadcast a new key block and a series of new in-between blocks at a higher height to reach decisions.

Proof: The leader starts a new view by collecting $2f + 1$ GENERIC messages and computing its $highQC$ before broadcasting a new GENERIC message. Suppose the highest kept lock among all replicas (including the leader itself) is $lockedQC = genericQC^*$. According to Theorem 1, the key block in Dolphin inherits the properties of the block in HotStuff, so there are at least $f + 1$ correct replicas have voted for previous key block that matches $genericQC^*$ and have sent it to the leader in their GENERIC messages. Therefore, the leader can learn a matching $genericQC^*$ from one of these messages and use it as $highQC$ in its new generic message. As all correct replicas are synchronized in their view and the leader is non-faulty, all correct replicas will vote in the new GENERIC phase. Once the leader assembles a valid $genericQC$ for this view, all replicas will vote for it. During this process, the replica that is about to become the leader keeps commit a series of in-between blocks to follow the same QC as the key block in the current view. Since these in-between blocks act as followers of the key block who do not require voting and do not affect the protocol's state machine, they will ultimately lead to new decisions along with the key block. After GST, the duration P for the phase to complete is of bounded length. □

Theorem 3: (Liveness) All correct replicas keep progressing consensus to commit new blocks in the correct view.

Proof: Under normal circumstances, an honest leader can obtain sufficient votes from other honest replicas to create a new block at a higher height as part of the standard protocol. However, a malicious leader may hinder progress by proposing alternate blocks at the same height, causing replicas to fail in committing previous blocks and reaching termination.

In the event of a malicious leader or significant network delay, the issuance of a new block at a higher height to commit previous blocks may fail. To ensure the protocol proceeds, each replica sets a local timer to wait for newly proposed blocks. If the waiting period exceeds the timeout, replicas switch to a new view to activate the next leader. Lemma 5 establishes that while the protocol context changes, the snapshot state remains the same. The new leader still employs the standard protocol to drive consensus. As long as an honest leader exists in subsequent views, it can always generate a valid new block at a higher height. □

Theorem 4: (Responsiveness) When network becomes synchronous, Dolphin can generate an honest leader to achieve consensus within the actual network delay, rather than the maximum delay.

Proof: Dolphin preserves the same responsiveness guarantees (i.e. Optimistically Responsive) as HotStuff because there is also no explicit "wait-for-Δ" step. It leverages the three-phase paradigm and logical disjunction in the SAFENODE function to override stale locks and switch to more up-to-date proposals. The leader can drive the consensus process forward without unnecessary delays, once $2f + 1$ votes are collected to form a new QC. □

For a more general explanation of the correctness, we give the following discussion: in-between blocks are included as additional chunks of information that are attached to the key blocks. The validity of an in-between block is inherently tied to that of the key block. If a malicious leader tries to take advantage of this, there are two potential strategies:

1) Sending the different in-between blocks to different replicas. In this case, due to the blockchain structure where each block's hash is dependent on the previous block, there will eventually be a difference (conflict) in the two key blocks proposed by the malicious leader. This situation then reduces to the case in HotStuff where a malicious leader proposes conflicting blocks in the same view. However, at most one of these conflicting key blocks can be valid because each key block requires $2f + 1$ replicas to vote, and even with f malicious replicas, having two valid key blocks in the same view would imply at least one correct replica voted twice, which is impossible. If a valid key block exists, it can gather a valid QC and be seen by correct replicas in subsequent views to update their states and chains. If not, then all the in-between blocks proposed by the same leader in the proposal's view will also be deemed invalid. Thus, safety and liveness are not violated in this scenario.
2) Sending the same in-between blocks but in a different order to different replicas. Suppose $A \leftarrow B \leftarrow Key$ (A, B are in-between blocks), but the malicious leader sends to replicas B before A. In this case, replicas cannot find B's direct parent, so subsequent blocks (including Key) are also invalid for them. If a key block exists in the current view with $2f + 1$ replicas accepting it, the situation is similar to case (1). If no such block exists, all blocks from the malicious leader in the proposal's view are deemed invalid, reducing to the case in HotStuff where the leader proposes an invalid block, again not violating safety or liveness.

Although all in-between blocks within a view have the same risk of being rejected as invalid when a malicious replica becomes the leader, resulting in a certain waste of bandwidth, this issue is common in leader-based BFT protocols, as the leader is entitled to propose blocks. A comparable issue arises in HotStuff. If a malicious leader continuously proposes invalid key blocks during its view, it will prevent consensus from being reached within that view, leading to a timeout and the waste of bandwidth for the entire duration of the view. Dolphin, however, ensures that when the leader is a non-faulty replica, it can achieve higher bandwidth utilization compared to HotStuff, as analyzed in Section VI-A.

VII. Evaluation

In this section, we first describe the implementation of Dolphin and the experimental setup. Then we present the results of our comparison of the performance of Dolphin and HotStuff in a wide area network (WAN) environment (the reason for choosing to conduct the experiment under the WAN environment is that compared to a local area network, WAN has a higher latency, which corresponds to our analysis), which includes various batch sizes, transaction sizes, leader switch frequencies, faults and network latencies. We also compare Dolphin with Narwhal, a novel DAG-based protocol, to demonstrate the efficiency of Dolphin in a WAN environment. Our main findings are that:

- Dolphin outperforms HotStuff in both latency and throughput in a WAN environment.
- Dolphin, as an instantiation of Asynchronization Procedure Patch, can bring performance comparable to that of DAG-based protocols to the original protocols.

We also evaluate Dolphin with existing HotStuff variants (WPBFT [19], FAST-HotStuff [22] and HotStuff with Coded Broadcast [23]) to illustrate the performance advantages of Dolphin over them in high-latency environments, the results are in the Appendix.

A. Implementation Details

We implement Dolphin and HotStuff in Rust, along with the Tokio library for asynchronous IO, and the Ed25519,[1] BLAKE3[2] for cryptographic operations. In our implementation, we use TCP connections for communication between nodes. Our implementation is composed of approximately 3500 lines of code, with only a minor difference of fewer than 100 lines between Dolphin and HotStuff, indicating the ease of implementation on existing protocols. Additionally, we implement a mechanism to limit the minimum number of transactions in in-between blocks to reduce the overhead. This mechanism allows Dolphin to make a regression to HotStuff when there are only a few transactions in the memory pool.

Our client is implemented as a separate single-threaded process that send transactions to nodes at a specified rate. Our nodes are able to read configuration from both JSON files and command line arguments, and output statistics as JSON files for further analysis. We also develop a module called CONFIG-GEN to automatically generate configuration files for different experiments, as well as scripts to distribute, run, and collect the results. Bash scripts are used to automate the execution of experiments, using the output from CONFIG-GEN and execute the experiments on multiple machines. All of the above code and measurement data is open sourced[3] on GitHub to enable reproducible results.

B. Setup

In this study, we conducted our experiments on a cluster of two Alibaba Cloud Elastic Compute Service (ECS) g6.3xlarge instances, each with 12 virtual CPUs (vCPUs) and 48 GB of memory, running Ubuntu 22.04 LTS. Two instances were located in Virginia, USA, and Sydney, Australia respectively, with an estimated round-trip time (RTT) of approximately 200 ms and a bandwidth of 50 Mbps between them.

[1] [Online]. Available: https://github.com/dalek-cryptography/ed25519-dalek
[2] [Online]. Available: https://github.com/BLAKE3-team/BLAKE3
[3] [Online]. Available: https://github.com/BerserkRugal/Dolphin/

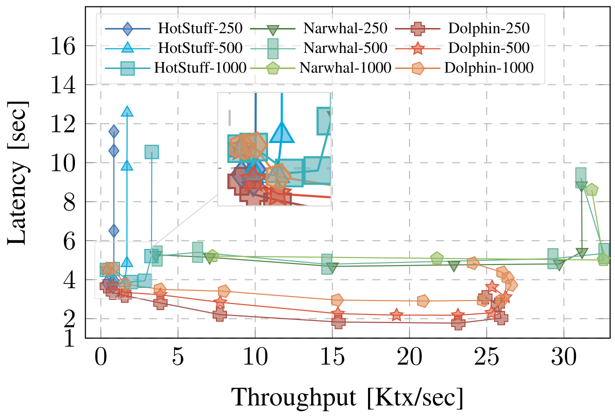


Fig. 7. Throughput vs. Latency with different batch sizes on a WAN.

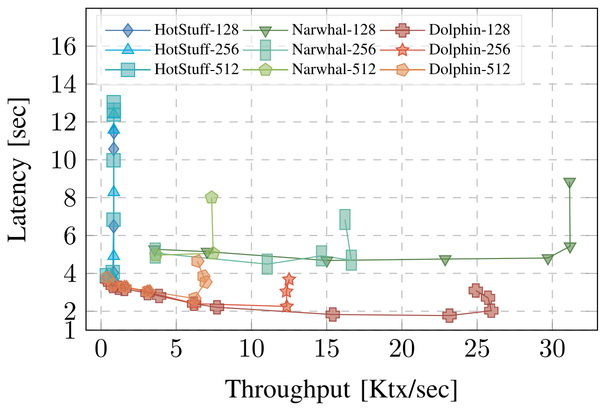


Fig. 8. Throughput vs. Latency with different payload sizes on a WAN.

For the experiments involving DAG-based protocols, we utilized Narwhal,[4] a novel DAG-based protocol designed for efficient performance in WAN environments.

In our evaluation of the performance of the protocols, we used latency and throughput as our main metrics. Latency was measured from the time a client generate a transaction to the time the transaction was committed by the protocol, and throughput was measured as the number of transactions committed per second. The results presented in this section were obtained by varying the sending rate of the client from underload to overload conditions.

End-to-end latency takes into account additional latency introduced by leader switches or faults, but it also includes other factors such as the implementation of the client and the routing policy of the message. These factors can make it challenging to discern patterns in the latency-throughput data, but we have included them in our results to provide as much information as possible.

In the following sections, unless otherwise specified, the batch size was set to 250 transactions per block, the transaction size was 128 bytes, leader switches were performed in a round-robin manner every five blocks for HotStuff and Dolphin, the number of nodes was four, the number of faults was zero, and the experiments were performed in a WAN environment.

C. Base Performance

The initial comparison is the throughput and latency of HotStuff, Narwhal, and Dolphin in a WAN environment with varying batch sizes, transaction sizes and committee sizes (number of nodes), as these parameters are commonly used to evaluate consensus protocols.

Fig. 7 shows the performance of three different batch sizes: 250, 500, and 1000, These batch sizes represent the maximum number of transactions in a block. All three protocols exhibit an L-shaped curve with similarities: the latency of the system is slightly higher when under load than under moderate load. However, once the system is fully loaded, the latency increases. Various batch sizes affect HotStuff's maximum throughput but do not affect the latency, since it cannot fully utilize the bandwidth of the WAN environment. Narwhal is not affected by the batch size, while Dolphin gets lower latency with a smaller batch size, due to the mechanism of the in-between blocks. At current batch sizes, Dolphin has lower latency (around 3 s) than Narwhal (around 5 s), but slightly inferior in peak throughput (at around 27,000 tx/s, slightly lower than Narwhal's 33,000 tx/s). However, when compared to HotStuff, which has a peak throughput of 3,300 tx/s and a latency of around 5 s (batch size of 1000), Dolphin has approximately 8 times the throughput and 40% less latency. When saturating the network, both Dolphin and Narwhal suffer regression, but this further proves that Dolphin can fully utilize the bandwidth of the WAN environment. In the zoomed-in area, we can see that Dolphin and HotStuff yield similar properties when under load. An overly saturated network is observed to cause some form of throughput regression. This is mainly caused by the underlying network structure, as we see both implementations (ours and Narwhal's) share similar results. The maximum throughput comparison between HotStuff variants and DAG-based protocols is not focused on in this paper, and we leave this optimization as future work.

Fig. 8 depicts the performance with three different payload sizes; 128, 256, and 512 bytes. The peak throughput of Dolphin and Narwhal in WAN is affected by payload size due to the bandwidth limit, while it is not for HotStuff. Dolphin's peak throughput (around 25,700 tx/s) is more than 25x higher than HotStuff's (around 860 tx/s) but slightly lower than Narwhal's (around 31,100 tx/s) under the same payload.

To demonstrate scalability, we assess the performance of the three protocols with committee sizes (number of replicas participating in the consensus) of 10, 50 and 100 replicas, as depicted in Fig. 9. To ensure that each replica is allocated at least one core, we used two DELL PowerEdge R750XS rackmount servers, each with 256 GiB of memory and 96 cores, located in Beijing, China to run this experiment and artificially simulate a 200 ms RTT and 50 Mbps bandwidth between them. The peak throughput we observe for Dolphin is 16,000 tx/s for a committee of 10 nodes, but lower for a larger committee of 50 nodes (around 9,000 tx/s) and even lower for a committee of 100, while Narwhal's peak throughput has been steady. HotStuff

[4][Online]. Available: https://github.com/asonnino/narwhal

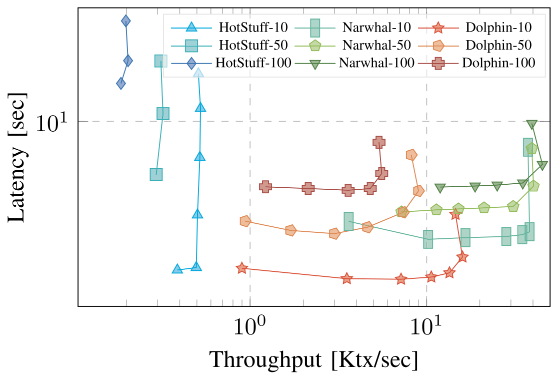


Fig. 9. Throughput vs. Latency with different committee sizes on a WAN.

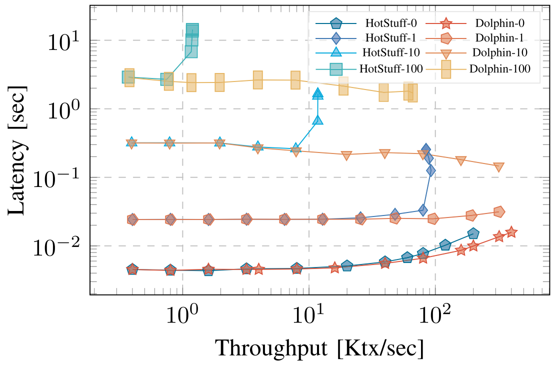


Fig. 10. Throughput vs. Latency with different network delays.

performs similarly to the Dolphin trend but with much worse performance. It peaks at only 530 tx/s for a committee of 10. Although chain-based protocols do not share the characteristic of having throughput unaffected by network size like DAG-based protocols, Dolphin's scalability is significantly improved compared to HotStuff through the Asynchronisation Procedure Patch.

These results show that Dolphin outperforms HotStuff in both latency and throughput in WAN. Our strategy effectively improves the bandwidth utilization of the original chain-based protocol and significantly reduces the performance gap with DAG-based protocols without introducing the graph structure.

D. Performance Under Different Network Delays

To further evaluate the impact on the performance of Dolphin and HotStuff, we vary the network delay between two nodes and draw the latency-throughput graph in Fig. 10. Since network delay under WAN is difficult to control manually, we start all nodes on the same server and communicate with each other through local ports in this experiment and implement a delay between nodes using `NetEm` provided by the Linux kernel. The default latency of `localhost` communication is around `0.01 ms`, which is negligible compared to WAN. We added `1 ms`, `10 ms`, and `100 ms` as comparison. Note that the results are presented in a log axis since they have multiplexer differences.

When the system is zero-loaded, both HotStuff and Dolphin have similar performance, as expected. However, when the system is saturated more, under the same throughput, Dolphin

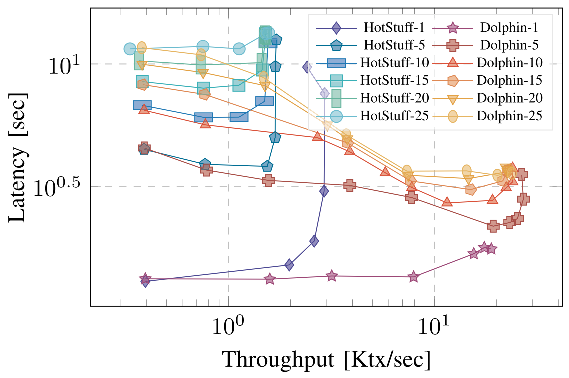


Fig. 11. Throughput vs. Latency with different choices of leader switch frequency on a WAN.

has lower latency than HotStuff. This is because, under the same client sending rate, Dolphin can consume many more transactions in a single view. After the saturation point, the throughput of HotStuff is constrained by latency, and this constraint becomes more and more critical when the network delay increases. With a network delay of 100 ms, HotStuff's throughput is only about 1,200 tx/s with a latency of nearly 15 s, while Dolphin's throughput still reaches 66,000 tx/s with a latency of no more than 3 s in the same case. These results also show that our method of introducing in-between blocks can help a protocol fully utilize the bandwidth of a WAN environment.

E. Performance Under Leader Changes

As we discussed in the analysis, the impact of leader changes on the performance of Dolphin can be significant. The frequent leader changes can block the generation of in-between blocks, as the next leader is yet in its view. To evaluate this impact, we varied the frequency of leader changes in every 1, 5, 10, 20, and 25 (key) blocks, and plotted the latency-throughput graph in Fig. 11. Please note that the latency difference of different leader switch frequencies is due to the backlog of transactions, which can be alleviated by introducing a transaction propagation or redirection mechanism. The results show that no matter how frequently the leader changes, Dolphin still outperforms HotStuff. Even when the leaders rotate every view, Dolphin's throughput of nearly 18,000 tx/s still reached more than five times that of HotStuff (3,000 tx/s). And when the leader changes less frequently, Dolphin can utilize more bandwidth in a WAN environment.

F. Performance Under Faults

We also evaluate the performance of HotStuff and Dolphin in the presence of faults. We simulate these faults by allowing certain nodes to deliberately drop all messages and not respond to any requests. In order to simulate the worst-case scenario, we not only set the number of faults f to be the maximum possible but also allow these faulty nodes to be the leaders. In our experiments, we set the total number of nodes $n = 4, 7, 10$, and the number of faulty nodes $f = 1, 2, 3$, respectively. The timeout of nodes is set to `7000 ms`, to prevent inconsistencies due to time differences and waiting for consecutive failed

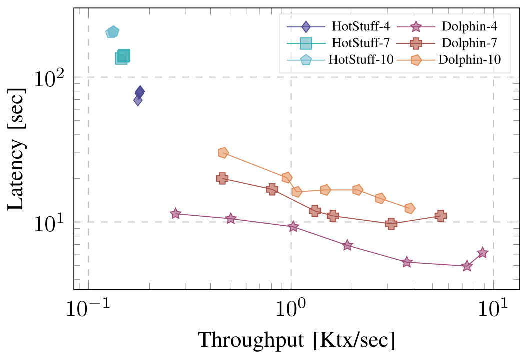


Fig. 12. Throughput vs. Latency under faults on a WAN.

nodes. We designate these faulty nodes as consecutive leaders in order to simulate the worst-case scenario that the system might face. In HotStuff's design, the pacemaker doubles the interval when the node does not make a decision in this period of time. This will cause the waiting time of the nodes to increase exponentially when encountering continuous faults. In practice, the committee can skip these apparently failed nodes through other mechanisms. The results are shown in Fig. 12. We can see that the timeout affects both HotStuff and Dolphin heavily in terms of latency and throughput. When a fault node is designated as the leader, the system performs poorly, and the performance becomes worse when consecutive leaders fail to respond. When a committee of 4 nodes suffers 1 faults, the throughput of HotStuff drops by over 5x (to around 180tx/s) and its latency increases by 11x (to around 77 s) compared to no faults. However, Dolphin is still capable of handling much more requests than HotStuff under faults, with only a 3x performance degradation (throughput to around 8,800tx/s, latency to around 6 s) compared to no faults.

VIII. Conclusion

In this paper, we propose Dolphin, a non-blocking BFT consensus protocol that addresses the performance issues through using concurrent block generation. We formalize the notion of Asynchronization Procedure Patch, a generalizable optimization technique that does not affect the execution process of the original state machine and can be applied with minimal modifications. The evaluation results show that Dolphin outperforms HotStuff in both latency and throughput in a WAN environment and maximizes network bandwidth utilization, similar to DAG-based protocols.

Based on the evaluation results, we observe that unrestricted generation of in-between blocks by the leader can sometimes saturate the network and result in throughput regression. While we have addressed this issue through the use of a minimal transaction ratio, further work is needed to optimize the threshold of in-between block generation. Another area of future work is the implementation of Asynchronization Procedure Patch in other novel BFT protocols, as well as the evaluation of their performance.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://assets.pubpub.org/d8wct41f/31611263538139.pdf

[2] M. Li et al., "Anonymous, secure, traceable, and efficient decentralized digital forensics," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 5, pp. 1874–1888, May 2024.

[3] M. Li, M. Zhang, L. Zhu, Z. Zhang, M. Conti, and M. Alazab, "Decentralized and privacy-preserving smart parking with secure repetition and full verifiability," *IEEE Trans. Mobile Comput.*, early access, May 07, 2024, doi: 10.1109/TMC.2024.3397687 .

[4] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," *IEEE Commun. Surv. Tut.*, vol. 22, no. 2, pp. 1432–1465, Secondquarter 2020.

[5] Y. Huang, J. Zhang, J. Duan, B. Xiao, F. Ye, and Y. Yang, "Resource allocation and consensus of blockchains in pervasive edge computing environments," *IEEE Trans. Mobile Comput.*, vol. 21, no. 9, pp. 3298–3311, Sep. 2022.

[6] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stak," *Self-Published Paper, August*, 2012. [Online]. Available: https://bitcoin.peryaudo.org/vendor/peercoin-paper.pdf

[7] Y. Du et al., "Blockchain-aided edge computing market: Smart contract and consensus mechanisms," *IEEE Trans. Mobile Comput.*, vol. 22, no. 6, pp. 3193–3208, Jun. 2023.

[8] D. Larimer, "Delegated proof-of-stake (dpos)," *Bitshare Documentation*, 2014. Accessed: May 17, 2024. [Online]. Available: https://how.bitshares.works/en/master/technology/dpos.html

[9] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[10] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[11] G. G. Gueta et al., "SBFT: A scalable and decentralized trust infrastructure," in *Proc. IEEE 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2019, pp. 568–580.

[12] V. Shoup, "Practical threshold signatures," in *Proc. Adv. Cryptol.–EUROCRYPT: Int. Conf. Theory Appl. Cryptographic Techn.*, 2000, pp. 207–220.

[13] C. Komlo and I. Goldberg, "Frost: Flexible round-optimized schnorr threshold signatures," in *Proc. Sel. Areas Cryptogr.: 27th Int. Conf.*, 2020, pp. 34–65.

[14] J. Kwon, "Tendermint: Consensus without mining," *Draft v. 0.6, Fall*, vol. 1, no. 11, pp. 1–11, 2014.

[15] V. Buterin and V. Griffith, "Casper the friendly finality gadget." *arXiv:1710.09437.*

[16] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2019, pp. 347–356.

[17] M. Baudet et al., "State machine replication in the libra blockchain," The Libra Assn., Tech. Rep., 2019. [Online]. Available: https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2019-06-28.pdf

[18] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 106–118.

[19] M. Gao, G. Lu, Z. Wang, and Y. Gao, "WPBFT: An improved consensus algorithm based on the hotstuff algorithm," in *Proc. IEEE 2nd Int. Conf. Artif. Intell. Blockchain Technol.*, 2023, pp. 56–59.

[20] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *Proc. Financial Cryptogr. Data Secur.: 26th Int. Conf.*, 2022, pp. 296–315.

[21] T. Cheng, "High performance consensus without duplication: Multi-pipeline hotstuff." *arXiv:2205.04179.*

[22] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai, "Fast-HotStuff: A fast and resilient HotStuff protocol." *arXiv:2010.11454.*

[23] I. Kaklamanis, L. Yang, and M. Alizadeh, "Poster: Coded broadcast for scalable leader-based bft consensus," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 3375–3377.

[24] X. Sui, S. Duan, and H. Zhang, "Marlin: Two-phase BFT with linearity," in *Proc. IEEE/IFIP 52nd Annu. Int. Conf. Dependable Syst. Netw.*, 2022, pp. 54–66.

[25] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: A dag-based mempool and efficient BFT consensus," in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 34–50.

[26] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *Proc. Adv. Cryptol.–EUROCRYPT: Int. Conf. Theory Appl. Cryptographic Techn.*, 2003, pp. 416–432.

[27] N. Papadis, S. Borst, A. Walid, M. Grissa, and L. Tassiulas, "Stochastic models and wide-area network measurements for blockchain design and analysis," in *Proc. IEEE INFOCOM -IEEE Conf. Comput. Commun.*, 2018, pp. 2546–2554.

[28] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, "On blockchain and its integration with IoT. challenges and opportunities," *Future Gener. Comput. Syst.*, vol. 88, pp. 173–190, 2018.

[29] A. Ggol, D. Leśniak, D. Straszak, and M. świtek, "Aleph: Efficient atomic broadcast in asynchronous networks with Byzantine nodes," in *Proc. 1st ACM Conf. Adv. Financial Technol.*, 2019, pp. 214–228.

[30] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2021, pp. 165–175.

[31] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 2705–2718.

[32] T. Jiang et al., "Non-blocking raft for high throughput IoT data," in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 1140–1152.

[33] J. Wang, D. Liu, X. Fu, F. Xiao, and C. Tian, "Dhash: Dynamic hash tables with non-blocking regular operations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3274–3290, Dec. 2022.

[34] Z. Zhang et al., "HCA: Hashchain-based consensus acceleration via re-voting," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 2, pp. 775–788, Mar./Apr. 2024.

[35] A. Stewart and E. Kokoris-Kogia, "GRANDPA: A Byzantine finality gadget." *arXiv:2007.01560.*

[36] I. Eyal, A. E. Gencer, E. G. Sirer, and R. V. Renesse, "{Bitcoin-NG}: A scalable blockchain protocol," in *Proc. 13th USENIX Symp. Networked Syst. Des. Implementation*, 2016, pp. 45–59.

[37] C. Li et al., "A decentralized blockchain with high throughput and fast confirmation," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 515–528.

[38] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–43, 2020.

[39] K. Hu, K. Guo, Q. Tang, Z. Zhang, H. Cheng, and Z. Zhao, "Leopard: Towards high throughput-preserving BFT for large-scale systems," in *Proc. IEEE 42nd Int. Conf. Distrib. Comput. Syst.*, 2022, pp. 157–167.

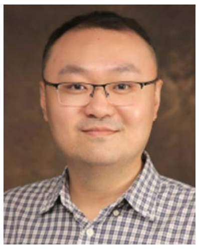
[40] F. Gai, J. Niu, I. Beschastnikh, C. Feng, and S. Wang, "Scaling blockchain consensus via a robust shared mempool." in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 530–543.

[41] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," in *Proc. Concurrency: Works Leslie Lamport*, 2019, pp. 203–226.

Xuyang Liu received the BE degree in computer science and technology from the Beijing Institute of Technology, in 2022. He is currently working toward the PhD degree with the School of Cyberspace Science and Technology, Beijing Institute of Technology, and the School of Computer Science, University of Auckland. His research interests include applied cryptography, blockchain technology, and distributed consensus.

Kaiyu Feng received the Bachelor of Engineering degree in computer science and technology from the Beijing Institute of Technology. He is currently working toward the master's degree with the School of Cyberspace Science and Technology, Beijing Institute of Technology. His research interests include blockchain technology, distributed systems, and databases.

Zijian Zhang (Senior Member, IEEE) is currently a professor with the School of Cyberspace Science and Technology, Beijing Institute of Technology. He is also a research fellow with the School of Computer Science, University of Auckland. He was a visiting scholar with the Computer Science and Engineering Department of the State University of New York at Buffalo in 2015. His research interests include design of authentication and key agreement protocol and analysis of entity behavior and preference.



Meng Li (Senior Member, IEEE) received the PhD degree in computer science and technology from the School of Computer Science and Technology, Beijing Institute of Technology (BIT), China, in 2019. He is currently an associate professor and dean assistant with the School of Computer Science and Information Engineering, Hefei University of Technology (HFUT), China. He is also a post-doc researcher with Department of Mathematics and HIT Center, University of Padua, Italy, where he is with the Security and PRIvacy Through Zeal (SPRITZ) research group led by Prof. Mauro Conti (IEEE Fellow). His research interests include data security, privacy preservation, applied cryptography, blockchain, TEE, and Internet of Vehicles. In this area, he has authored or coauthored 77 papers in international peer-reviewed journals and conferences, including *IEEE Transactions on Information Forensics and Security*, *IEEE Transactions on Dependable and Secure Computing*, *IEEE/ACM Transactions on Networking*, *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Knowledge and Data Engineering*, TODS, *IEEE Transactions on Services Computing*, *IEEE Communications Surveys and Tutorials*, ISSTA, MobiCom, ICICS, SecureComm, and TrustCom. He is a Senior Member of CIE, CIC, and CCF. He is an Associate Editor for *IEEE Transactions on Information Forensics and Security*, *IEEE Transactions on Network and Service Management*, and *IEEE Internet of Things Journal*. He was sponsored by ERCIM 'Alain Bensoussan' Fellowship Programme (from 2020.10.1 to 2021.3.31) to conduct Post-Doc research supervised by Prof. Fabio Martinelli at CNR, Italy. He was sponsored by China Scholarship Council (CSC) (from 2017.9.1 to 2018.8.31) for joint Ph.D. study supervised by Prof. Xiaodong Lin (IEEE Fellow) in the Broadband Communications Research (BBCR) Lab at University of Waterloo and Wilfrid Laurier University, Canada.

Xi Chen received the BE degree in software engineering from Dalian Maritime University, in 2021. He is currently working toward the PhD degree with the School of Cyberspace Science and Technology, Beijing Institute of Technology. His research interests include blockchain technology and distributed consensus.

Wenqian Lai is currently working toward the Undergraduation degree with the School of Cyberspace Science and Technology, Beijing Institute of Technology. His research interests include include blockchain technology and artificial intelligence security.

Liehuang Zhu (Senior Member, IEEE) is currently a professor with the School of Cyberspace Science and Technology, Beijing Institute of Technology. He is selected into the Program for New Century Excellent Talents in University from Ministry of Education, P. R. China. His research interests include cryptographic algorithms and secure protocols, Internet of Things security, cloud computing security, Big Data privacy, mobile and Internet security, and trusted computing.