Multi-pipeline HotStuff: A High Performance Consensus for Permissioned Blockchain

Taining Cheng*, Wei Zhou*, Shaowen Yao†, Libo Feng*, and Jing He†
Yunnan University*, China
Email: tirning@outlook.com, {zwei, yaosw, fenglibo, hejing}@ynu.edu.cn

Abstract—The state-of-the-art HotStuff operates an efficient pipeline in which a stable leader drives decisions with linear communication. However, with the unifying proposing-voting pattern, it takes two rounds of messages to produce a certified proposal, which severely limits the performance of the consensus protocol and makes it difficult for the blockchain to exert the bandwidth and concurrency of modern operating systems. Thus, this paper developed a new consensus protocol, called Multi-pipeline HotStuff, for permissioned blockchain. To the best of the authors’ knowledge, this is the first protocol that combines multiple pipelines of HotStuff to propose batches in order, such that proposals are built optimistically when a correct replica realizes that the current proposal is valid and will be certified by quorum votes in the near future. Simultaneous proposing and voting allow the protocol to produce more proposals in every two rounds of messages, it further boosts the throughput at a comparable latency with that of HotStuff. The evaluation experiment confirmed that the throughput of the proposed protocol outperformed HotStuff by approximately 60% without significantly increasing end-to-end latency under varying system sizes. Even if the protocol frequently performs view-change phase due to network asynchrony, its optimization continues to demonstrate better performance.

Index Terms—Blockchain, Consensus, Byzantine fault tolerant, Partial synchrony

I. INTRODUCTION

The emergence of blockchain has significantly affected the study of Byzantine fault-tolerant (BFT) consensus in the last decade. As a core infrastructure of blockchain, the BFT consensus protocol [1], [2] build the basis for implementation of state machine replica and the execution of smart contracts in the decentralized network. Through the BFT protocol, each party of the system maintains the same state, even enduring arbitrary failures. The protocol ensures that $2f + 1$ out of $n$ correct parties agree on the order of requests (where $n$ is system size and $f$ is the threshold of failures, such that $n > 3f$). To reach agreement on the order of requests, the state-of-the-art BFT protocol HotStuff [3] employs a leader-based method to drive consensus instances to identically commit commands or transaction for maintaining state under partial synchrony.

With threshold signature schema [4], the communication cost of consensus decision was amortized to linear. HotStuff acquires the first two phases to lock a safe position that will never be forked and an additional phase to commit the requests in that position. Further, a chained structure was introduced to combine the results of voting into next proposal. This structure unifies three phases of one proposal into a proposing-voting pattern, effectively improving system performance with the features of linear cost.

Although the chained structure opened a new chapter that bridged the classical BFT and blockchain consensus. A performance bottleneck gradually surfaced. In the context of this linear proposing-voting pattern, it is referred to as a single-pipeline bottleneck, which describes the poor efficiency when it is used as a critical commit path in the consensus protocol. From a macro perspective, regardless of the role, either a proposal drives replicas to validate the proposal and send a vote, or quorum votes drives a new leader to propose. The current manner in which driven messages are produced and transmitted among replicas leads to the following performance bottlenecks:

- **Idle time.** A simple proposing-voting pipeline results in the replica always facing a period of idle waiting time that requires no action, owing to no driven message being received. With the system operation. The accumulation of idle time limits the performance of a system that cannot facilitate execution.
- **Conservative extending.** The pipelined operation provides the ability to extend an on-consensus blocks that have been certified but not yet committed. However, the conservative extending costs two rounds of messages to generate a block, which limits a degree of performance gains.
- **Poor concurrency.** Every replica requires considerable time to perform cryptographic primitives, each of which is computationally intensive. Meanwhile, the transmission module is idle when the replica performs computation on the CPU, thereby resulting in the concurrent capability of the multi-core system and the bandwidth not being fully exerted.

The natural intuition for optimizing the single pipeline pattern involves each replica producing the maximum proposals possible to fully drain the bandwidth and the ability of
concurrent [5], [6]. They allowing parallel leaders to propose batches independently and concurrently. This led to improved performance, despite the need for an extra transaction partition strategy to preclude duplication problems. While the dilemma is that these duplications can simply be filtered out after an ordering, but the damage has already been done—excessive resources, bandwidth, and possibly CPU have been consumed. Consequently, a natural question arises: Is it possible to design a consensus protocol that avoids duplications while breaking the performance bottlenecks in the single pipeline design?

Contributions. To our knowledge, we present Multi-pipeline HotStuff (MPH), the first novel BFT protocol that ensures one proposal (block) is generated per round of messages. Without increasing the latency, MPH produces twice as many proposals as those produced by HotStuff owing to the redesigning of the consensus instance. Moreover, MPH provides efficient view-change and responsiveness [3] through the adoption of a three-phase proposing-voting pattern in each pipeline [7].

The key idea involved in MPH is that when the correct leader realizes that the latest block can be voted (verified), it optimistically proposes by extending that verified one. This optimistic proposing resolved several of the challenges mentioned above. First, simultaneous proposing and voting can facilitate the concurrency of modern operating system. Since the computationally intensive cryptographic operations and network transmission are fully exploited. Second, the optimistic proposing by the correct leader reduces the two rounds of delay to one. The period of idle time in the single pipeline owing to no driven messages being received in the single pipeline is fully utilized. Third, MPH does not suffer from the duplication issue because it does not adopt parallel leaders to address the performance bottleneck.

Another feature of MPH is that the costs for a new leader to drive consensus after view-change is quadratic. With Optimistic proposing, uncommitted blocks from multiple pipelines will be considered as unsafe position to extend in the event of a network failure or the malicious leader goes silent. The view-change of MPH achieves safety by adding multiple proofs of latest certified block to synchronize the global view of certification, so that those unsafe blocks are forked and a new block extends at least from the certified position. Then, all the locked blocks are eventually committed after the GST. As such, we expect this small price of added proofs to messages in return for considerably improving the overall performance.

The remainder of this paper is organized as follows. The related work is introduced in Section II. Section III describes the system model and preliminaries. Section IV presents an analysis of the HotStuff, and explores the factors contributing to performance metrics. Sections V and VI provide the design and implementation of the protocol, respectively. Further, the evaluation are presented in Sections VII. The conclusions and future works are presented in Section VIII.

II. RELATED WORK

Reaching consensus in the presence of Byzantine failures was abstracted as Byzantine Generals Problem by Lamport et al. [2], who also introduced the term “Byzantine Faults”. The first theoretical synchronization solution in a fault-tolerant system was provided by Pease et al. [8], wherein an exponential number of messages was sent to reach an agreement. Thereafter, a practical polynomial algorithm was proposed by Dolev, and Strong [9], with the tolerance of \( f < n \). To further improve efficiency, the randomized method was introduced [10]–[12]. Kutz and Koo [11] proposed an expected constant-round and \( f < n/2 \) resilience protocol using moderated Verifiable Secret Sharing (VSS) and later improved by the protocol with quadratic communication cost [12]. Recent studies of Abraham et al., Sync HotStuff [13] have shown a practical BFT solution with a latency of \( 2\Delta \) (where \( \Delta \) is the upper bound of message delay) and quadratic communication. Another study [14] allowed optimistic responsive leader rotation to shrink Sync HotStaff’s latency upper and lower bounds.

The partial synchronous model was first proposed by Dwork, Lynch, and Stockmeyer [15], who also first proposed a protocol that costing \( O(n^3) \) communication and \( O(\alpha) \) round per decision. However, expensive costs resulted in the protocols remaining theoretical until PBFT was devised by Castro and Liskov [16]. PBFT reaches consensus by reliable broadcast with \( O(n^2) \) communication complexity and two round-trips per decision. With quadratic cost in the normal and view-change cases being referred to as Tendermint [17]. Kotla et al., introduced Zyzzyva [18], which considered that an optimistic path has a linear cost, while the view-change path remained \( O(n^3) \). However, a safety violation was found by Abraham et al., who also presented revised protocols [19]. SBFT [20] was shown to address the challenge of scalability based threshold signatures and an optimistic path. Along with the threshold signature scheme and pipelining, the linear cost and optimistic responsive HotStuff [21] opens a new picture of modern consensus protocol. Further, the DimBFT [22] from Meta and Fast-Hotstuff [23] refined HotStuff as a core of the blockchain system. It retained linear costs and introduced an explicit liveness mechanism that prevents unnecessary delays. Inspired by past studies, Chan et al. described an extremely simple and natural paradigm with quadratic cost referred to as Streamlet, [24], which finalized a proposal when it was extended by another two certified proposal.

Moreover, FLP [25] theory stated that reaching deterministic consensus is impossible in the face of even a single failure. Fortunately, the impossibility was circumvented by harnessing randomness; several studies have focused on asynchronous settings, including the first practical asynchronous protocol given by Cachin et al. [26], HoneyBadgerBFT [6], VABA [27], Dumbo-BFT [28], and DAG-Rider [29]. Most of these protocols have over-quadratic costs but are always alive even in asynchrony.
III. Preliminaries

This paper assumed a system comprising a fixed set of $n = 3f + 1$ replicas, each replica indexed by number $i$, where $i \in \{1, ..., n\}$. Replicas corrupted up to $f$ of the total by the adversary are referred to as Byzantine faulty and may arbitrarily deviate from the protocol, while the rest of the replicas are correct.

A. Communication and Network

As a distributed system, Network communication links are reliable and authenticated. All messages sent among correct replicas are eventually delivered if and only if the sender sent that message to the receiver. However, the adversary can control the links by controlling the delivery time. This paper referred to the network assumption—partially synchronous proposed by Dwork et al. [15] to model the links, where there is a known bound $\Delta$ and an unknown Global Stabilization Time (GST).

B. Cryptographic Assumptions

A standard digital signature was assumed, and public-key infrastructure (PKI) [4], [20], [30] was provided by a trusted dealer. All replicas were equipped with a single public key, and each of the $n$ replicas was assigned a distinct private key. A replica $i$ can use its private key to sign a partial signature on message $m$, which is denoted by $p_i \leftarrow \text{t}sign_i(m)$. The $(k, n)$-threshold signature scheme, a set of partial signatures $P = \{p_i | p_i \leftarrow \text{t}sign_i(m), i \in I, |I| = k\}$ of any $k$ replicas can be used to generate a digital signature $\sigma \leftarrow \text{tc}ombine(m, P)$ on message $m$, where $I$ is the set of replica indexes that contributed partial signatures. Further, a replica is aware that $n$ replicas have given valid partial signature on message $m$ if the function of $\text{tverify}^\gamma(m, \sigma)$ return true. Generally, given a valid signature $\sigma$ on $m$, no adversary or adversaries has an overwhelming probability of generating a signature $\sigma$ for $m$. In the context of distributed setting, this paper adopted the threshold of $k = 2f + 1$.

In addition, a cryptographic hash function $h(\cdot)$ [31], which mapped an arbitrary length input to output of fixed size was assumed. The probability of any replica generating a couple of distinct inputs $m$ and $m'$ subject to $h(m) = h(m')$ is negligible; and the output of the hash function can be used as a unique identifier for input.

C. BFT SMR

In the distributed context, the consistency of the state among all replicas is obtained through State Machine Replica [1], [16], wherein each replica executes transactions in the same order. The BFT protocol aids in each correct replica safely committing transactions in a total order. It is expected that the protocol never compromises the safety and liveness:

- **Safety**: All correct replicas commit the same transaction at the same log position.
- **Liveness**: Each transaction from client is eventually committed by all correct replicas.

The BFT SMR problem solved by HotStuff [21] allows for pipelining of decision. This paper also followed the path to solving the BFT SMR problem, with further details presented in section 4.

D. Notations

First, the notations used throughout the whole protocol are elucidated.

**View Number.** The BFT protocol is implemented via many instance of single-shot Byzantine agreement. Each replica indexes the instance by the view number, which is initially to 0 and then increases monotonically.

**Block structure.** The block is denoted by a tuple $b = \langle id, v, p, txs, qc, \rho \rangle$, where $id = \text{hash}(v, p, txs, qc)$ is the digest of the current block’s information. $v$ is the view number of block; each block must refer to the block id of the previous view by $\rho$ to indicate which block the current block is extended to. Further, $txs$ is a batch of pending transactions, $\rho$ is the partial signature from the proposer on block’s id, and $qc$ is also defined below.

**Quorum Certificate.** A Quorum Certificate(QC) is formatted as a tuple $qc = \langle block, v, \sigma \rangle$. It is a data type for proof where the block $B$ is validly signed on block id, generated via the combination of the partial signature shares from a quorum $n - f = 2f + 1$ replicas. A QC implies that a block is received by a quorum replica and certified. Given a data structure of $x$, $x.y$ was used to refer to the element $y$ of the original structure $x$ throughout this paper. In addition, the blocks and QCs can be ordered based on their view number.

**Timeout Certificate.** A timeout message of view $v$ contains a partial signature share on $v$, and the highest QC $q_{ch}$ of the sender. A timeout certificate (TC) was used to prove that the protocol requires another leader to drive consensus, it is generated through the combination of shares of $n - f = 2f + 1$ timeout message. A valid TC must contain a set of QCs with view number $< tc.v$ denoted by $tc = \langle v, q_{ch}, \sigma \rangle$.

E. Complexity Measurement

We measure the complexity of protocol using the number of authenticators instead of message size, because the bit of message cannot be bounded well to capture the cost. Every message must be signed by the sender, and the receiver must verify the signature and combine quorum partial signature shares; these cryptographic operations are computationally intensive. Thus, the authenticator complexity is a suitable way to capture the overall costs of the protocol.

IV. DESCRIPTION OF PIPELINE PARADIGM

There are two components of the HotStuff, linear normal commit progression and quadratic cost view-change. During the normal progression, a designated leader $L$ proposes a block $b$ that extends the block associated with the highest QC of its own. Thereafter, other replicas attempt to update their locked and the highest QC to check if any block can be committed when receiving block $b$. Then, block $b$ is voted for by sending a partial signature share to the next leader.
forms a QC for view \( v \) when it collects \( n - f \) votes of the previous view \( v \) and enter view \( v + 1 \). The normal case repeatedly drives the system to commit transactions linearly. However, in case of any timeout owing to delay of the transmission or a malicious leader remaining silent, the timeout message being broadcast in view \( v \) must carry a highest QC of the sender. This allows the new leader possessing the ability to prove that no action more recent than the block of that QC is committed. The leader generates a TC and proposes a block with it, where the proposed block should extend the block with the highest QC in the timeout messages. When any replica receives a block with a TC, it attempts to transfer to the next view and checks if any block satisfies the 3-chain commit rule, which implies that the first of three adjacent certified blocks can be committed.

**Vote Rules.** Before a replica vote for the proposal, whether at least one of two rules is satisfied is examined as follows:

- \( b.v = b.qc.v + 1 \)
- \( b.v = b.tc.v + 1 \) and \( b.qc.v \geq qc_{\text{locked}.v} \)

Block \( b \) contains either a QC or a TC. The first rule implies that voting for a block with monotonically increasing view numbers and directly extend the block of \( b.qc.block \) is safe, owing to no possibility of two QCs forming in one view. The second rule implies that a valid block must extend at least a locked block, which is \( qc_{\text{locked}.block} \).

### A. Throughput and Latency Analysis

Considering the study on dissecting the chained-BFT performance [32], the life-cycle of the manner in which a block can be certified was analyzed to obtain the correlation between the permanence metrics and the vital environment factors. A consensus instance comprises proposing and voting steps, where the time consumption can be used to calculate the throughput and latency.

Here, the focus is on the synchronous network, while Byzantine faults are ignored. The average service time that a block is certified can be divided into several parts:

\[
t_s = 3t_{\text{CPU}} + 2t_{\text{NIC}} + t_L + t_Q
\]

where \( t_{\text{CPU}} \) captures the delay of signature operations (can be a constant number), \( t_{\text{NIC}} = 2mb^{-1} \) is the delay of data frame conversion at sender and receiver, where \( m \) is the total size of a block, \( t_L \) is the Round-Trip Time [33] that captures the network transmission delay by the normal distribution, and \( t_Q \) is the expected delay of collecting a quorum of votes from replicas, which is modeled by \( \left( \frac{2^N}{N} - 1 \right) \) order statistics of \( N - 1 \). Recall that, each new certified block commits an ancestor certified block, and \( t_s \) is used to approximately compute TPS, which is denoted as \( TPS \approx |m|/t_s \). Further, the BPS (Bytes Per Second) can be obtained as

\[
BPS \approx \frac{1}{(3t_{\text{CPU}} + 2t_Q + t_L) \cdot m^{-1} + 4 \cdot b^{-1}}
\]

However, this is far from reaching the upper limit of bandwidth despite the first term of \( (3t_{\text{CPU}} + 2t_Q + t_L) \) being close to 0. Thus, regardless of \( b \) value, the overall \( BPS \) provided by any distributed cooperative system is much smaller than the bandwidth, particularly when \( 3t_{\text{CPU}} + 2t_Q + t_L \) contributes in excess. In addition, the average latency can be modeled as

\[
Latency \approx 3t_s + 6t_L
\]

### B. Limitations

These Analysis provide a quantitative methodology to show which part of time consumption is contributing to the total block service time within one instance of pipeline method. The term \( t_{\text{CPU}} \) and \( 2t_{\text{NIC}} + t_L \) occurs in sequence, resulting in the same period of time \( \Delta = 3t_{\text{CPU}} + 2t_{\text{NIC}} + t_L \), with either only CPU or network contributing to the actual block service time. First, it is embodied in the consensus that only one block is to be proposed in the two rounds of messages. Second, the period of time \( 2t_{\text{NIC}} + t_L \) is only used for sending or receiving messages and not for creating blocks or preparing votes simultaneously. In particular, the term of network delay significantly exceeds the term of CPU. However, the design of the single-pipe protocol itself becomes a performance bottleneck that is difficult to breakthrough.

As shown in Figure 1, the time required for both voting and proposing round during the running of implementation of the standard HotStuff was measured; the consumption of two steps is approximately equal except for abnormal data. In other words, the parallelism of the single pipeline is low, as replicas are required to wait for another round of voting after proposing, thereby resulting in a lower block generation rate. Thus, the next section shows that it is possible to better utilize each round by shrinking two rounds of the view into one round to exploit each round. The validator votes in each round, whereas the leader proposes in each round.

![Figure 1](image-url)

**Fig. 1.** Time consumption of proposing and voting phase within one view

### V. Multi-pipeline Hotstuff Design

To strengthen the performance of this pipeline method, we propose a multi-pipeline scheme that can make the replica proposing and voting in every message round. We call the new protocol Multi-pipeline Hotstuff (MPH), which has linear communication cost for normal phase and quadratic cost for view-change phase. Although theoretically cost is the same as Hotstuff, the performance is almost doubled. In this paper, we only take two pipelines into consideration and leave exploration to future work, the full picture of protocol is presented in algorithm 1.

**Protocol intuition** The idea behind our multi-pipeline is that when the correct replica considers a block to be valid, then
in the optimistic case, that block will be certified by quorum votes in the future. Thus, the correct leader optimistically proposes a block that extends the latest verified block, as well as the validators vote for that verified block. In other words, voting and proposing happen at the same time but for different pipeline. Such as view $v + 2$, the leader of $v + 2$ proposes a block $b + 2$, the validators vote for block in view $v + 1$. As we can see from Figure 2, each view contains only one round of message exchange for voting and proposing; more blocks will be proposed in the same number of rounds compared to the implementation of single pipeline.

Fig. 2. Overview of Multi-pipeline Hotstuff

A. Complementary of model

**Timeout Message.** The timeout message is used to form a complete TC, which is denoted by a tuple $tm = \langle vf, vs, p >$. Here $vf$ is the current view that time outs and is used to form a TC that transfers to the first view after view-change. Further, $vs$ is for the next view of $vf$; and $p$ is the partial signature shared on $vs$ and $vf$.

**Block structure.** The block structure is defined in multi-pipeline pattern as $b = \langle id, v, p, pv, txs, qc, tcs, tc, \rho >$, where $tc$ and $tcs$ are current formed by replicas, $tc$ is used by those lagging replicas and $tcs$ is a set of TC, which will be used to verify the $tc$ is valid. Further, $pv$ is the view of block with id $p$ and can be used to check safety rule when time outs. Note that, each block contains two "links" to previous block; one is block’s id of the parent and indicates the global order of block. And the other is the QC of the previous block, which indicates that the previous block was voted by quorum replicas.

B. Block State

Two are two types of block states during the consensus process, the **certified** and **verified** states, were defined. A block is considered certified if there exists a QC for the block, that is, $qc.block = b$. Further, a block is verified if the proposer’s partial signature was authenticated and the QC in the block was well-formed by a quorum. A correct replica will perform the proposing and voting step based the current block state. The new certified block will commit a preceding certified block, and the new verified block will be extended to create a new proposal for next view.

C. One Round Trip View

Each view has a designated leader, which is deterministically defined by the view number (e.g., $id = v \mod n$). The difference lies in the fact that there is only one round message trip occurs within a view. As soon as the replicas enter a new view, it immediately votes and proposes (if it is leader of next view) after the current block is verified. A block state transition to verified implies that voting and proposing can occur in parallel at an optimistic point. Owing to proposing and voting are parallelized in different pipeline, it is also necessary to ensure view is contiguous. Consequently, the view number of the block’s QC is computed by $qc.v = b.v + 1$.

D. 3-chain Predicate in M-Hotstuff

Each pipeline of MPH performs in terms of the 3-chain rule, the protocol tracks certain vital variables of block and QC. The first of two adjacent certified blocks is referred to as the **locked** block, which indicates that no higher block could have reached a commit stage. Further, the QC corresponding to the block is referred to as locked QC. In this paper, two types of locked variables were defined: $qc_{cur.lock}$ used to keep track of the current locked position to which pipeline the currently processing block belongs, and $qc_{lag.lock}$ used to keep track of the previous locked position where another pipeline exists. Because there are multiple locked positions in the multi-pipeline setting, a block is expected to extend at least the locked block, which is the block certified by $qc_{cur.lock}$. In addition, another two variables record the highest and the second-highest QC, denoted as $qc_{high}$ and $qc_{sec.high}$, respectively. These are used to form a new chained block for their respective pipelines.

E. Normal-Case Operation

When a replica realizes that it will act as the leader in view $v$, it will collect votes for the block in the previous view $v - 1$ to generate the QC (suppose $qc$). Thereafter, a block $b$ that extends the latest verified block will be created by the leader once the leader enters view $v$ by processing QC $qc$. Especially, a digest of extended block and $qc_{high}$ ($qc_{sec.high}$ has been updated to $qc$) must be embedded in the block as proof to relay to other replicas for ensuring safety.

As a validator, when receiving the first block of view $v$ from $L$, any replica attempt to verify the authenticity of the leader’s
proposal \( b \) to check if the proposer is the leader of the block’s view and if the signature on the block is valid. Further, if the QC in block \( b \) is valid, then the block associated with QC becomes certified, that is, \( qc_block \). Subsequently, the block’s state becomes verified. Validators increase its view number to enter view \( v \), and the second-highest QC is updated to the highest QC, followed by the highest QC to the latest QC in block \( b \). The \( qc_{laslock} \) and \( qc_{curlock} \) are also updated one step forward. Thus, batch transactions in the first block of three adjacent certified blocks are chained by QC (new QC forms another three adjacent certified blocks), which are provided to the execution module to commit. Optimistically, the block that is considered verified will eventually be certified in the next view unless the next leader is faulty. Based on this, two parallel sub-procedure were designed for simultaneous proposing and voting. If the validator is the next leader, it sends a signal to the proposing procedure to start creating a proposal for that view, provided there is a valid QC of the previous view generated. Meanwhile, the validator checks whether this proposal can be voted by the safety rule. According to the 3-chain predicate defined above, the safety rule for voting in the normal case is obtained as:

- **Safety**: \( b.v = b.qc.v + 1 \) and \( b.pv = verifiedB.v \), the block \( b \) directly extends from the latest verified block with latest QC.

If the proposal is satisfied with the safety rule, then a threshold signature on \( b \) is sent to the leader of view \( b.v + 2 \) as a vote. Finally, the normal case repeatedly drives the system to decide the order of transactions.

**F. View-Change Case**

When the timer of a certain view \( v \) expires, owing to the lack of a mechanism to detect whether the timeout is caused by asynchrony, the message is delayed, or the leader is faulty. The view-change mechanism is triggered to skip to the next view \( v + 1 \) and replace another leader for ensuring liveness. In the context of this paper, two blocks were considered uncertified. Consequently, the view-change period of MPH comprised two consecutive views, each of which required a corresponding TC to be advanced. As the timeout occurs, all replicas stop voting for that view and multicas a timeout message containing a threshold signature share for \( v - 1, v \) and its highest QC \( q_{sechigh} \) as well as second-highest QC \( q_{sechigh} \). When any replicas receive a quorum of such timeout messages. First, the timeout messages are checked to keep having the latest QC. Thereafter, two TCs of view \( v \) and \( v + 1 \) are formed through the combination of quorum partial signature shares, denoted as \( tc1 \) and \( tc2 \), respectively. Finally, the liveness rule for voting can be expressed as:

- **Liveness1**: \( b.v = b.te.v + 1 \) and \( b.qc.v = qc_{laslock}.v + i * 2, b.pv = qc_{laslock}.v - 1 + i * 2, i \in \{0, 1\} \),
- **Liveness2**: \( b'.v = b'.te.v + 1 \) and \( b'.qc.v = qc_{laslock}.v + i * 2, i \in \{0, 1\} \) and \( b'.pv = verifiedB.v \).

In the first view, after timeout, the leader of that view \( v + 1 \) first enters through TC \( tc1 \), and then proposes a block that directly extends the block of \( q_{sechigh} \). Recall that the structure of block defined in Section 5, the two TCs and QC \( q_{sechigh} \) must be included inside the block. Once the validators receive the block from the leader, it updates its own view based on TC and checks the cryptographic semantics of the block. Further, it computes whether the block is a valid branch and changes the verified branch to the new block. The Liveness1 rule states that a valid proposal in the first timeout view must contain a QC that is at least as new as \( qc_{laslock} \), and the block must directly extend from the successor block of \( b.qc.block \). If Liveness1 is satisfied, then the same parallel sub-procedure is repeated; the next leader first updates its view through second TC \( tc2 \), and the vote of \( b.v + 2 \) is sent to the leader of view \( b.v + 2 \). When the leader reaches the new view \( b.v + 2 \), it creates a block \( b' \) that directly extends from the latest verified block, where the block also contains two TCs and the latest QC \( q_{sechigh} \). In the second timeout view, the validators also update the view number by TC \( tc2 \) and check the authenticity of the proposer of block \( b' \), using the Liveness2 predicate to determine whether to accept the block. Here, the QC \( b'.qc \) cannot be higher than the QC \( qc_{laslock} \), and \( b' \) must be a successor of latest verified block. Consequently, all replicas vote for \( b' \).

As now a view-change process after the timeout has finished, it is guaranteed that the system will attempt to keep working in the event of timeout due to asynchrony or a faulty leader problem. Thus, it first needs to synchronize the view such that at least a quorum replicas start with the same view to prevent inconsistency problems caused by out-of-sync blocks. Regardless of normal or view-change cases, the voting predicate is true provided either one of the safety and liveness rules holds. Thus, any replica can use it to determine whether to vote for the proposal.

**G. Complexity and Latency**

Obviously, when the network is synchronous, and the leader is honest, a complete threshold signature of size \( O(1) \) in QC needs at least quorum partial signature shares. Every share of size \( O(1) \) is given by a unique replica’s cryptographic tool as a vote. Owing to a constant number of authenticators in each block, it can be concluded that the overall complexity of reaching a consensus decision in a normal case is \( O(n) \). A block adopts the 3-chain to be committed, and one more round for all replicas receives the certification. This results in a latency of 7 rounds. In the view-change case, all replicas need broadcast timeout messages with QC for view synchronization, and each message has size \( O(2) \), and the subsequent view-change message of the leader’s proposal contains three authenticators \( O(3) \), corresponding to one QC and two TCs. The responding vote also has a single share of size \( O(1) \). Consequently, the view-change case yields \( O(4n^2) \) and \( O(3n) \) cost of view synchronization and view-change, respectively. For latency, a locked transaction that suffered asynchrony with a latency of at least nine rounds, if the network returns to synchronous. The branch costs two more extra rounds.
However, the round of latency goes to an infinite round if the GST never occurs.

VI. IMPLEMENTATION

This paper implemented all protocols discussed: HotStuff, Streamlet and MPH in Golang. The TCP\(^1\) was used to build reliable point-to-point channels to implement the SMR-BFT abstractions correctly. Further, secp256k1\(^2\) was used for elliptic curve based signatures. The implementation of MPH\(^3\), HotStuff and the corresponding single pattern protocol were open-sourced.

In addition, the goal of the protocol is to reach an agreement on the order of transactions instead of the content itself. In the implementation, a subsystem of mempool (transaction buffer) is built, which removes the transaction dissemination from the critical path of consensus. Therefore, the transaction body need not be included in consensus blocks. The protocol only agrees on the digest of these transactions, implying that all the replicas’ mempool is shared. This significant optimization has been proven by Narwhal [34], which can further improve performance compared to other studies that have not equipped the sharing strategy.

Algorithm 1 Main loop of MPH for replica \(i\)

1. \(\text{initialize} :\)
2. \(\text{timer.Reset}()\)
3. \(\text{currView} \leftarrow 1\)
4. \(\text{Send newView} M_{\text{newView}}\) to procedure Proposing
   ▷ Normal phase
5. \(\text{procedure BLOCKPROCESS:}\)
6. \(\text{while} \ True\ do:\) wait event \(M\)
7. \(\text{if} \ M\ is a proposal message then\) ProposeProposal\((M)\)
8. \(\text{procedure PROPOSING:}\)
9. \(\text{while} \ True\ do:\) wait event \(M\)
10. \(\text{if} \ M\ is a newView message then\) Propose\((M)\)
11. \(\text{procedure VOTING:}\)
12. \(\text{while} \ True\ do:\) wait event \(M\)
13. \(\text{if} \ M\ is a vote message then\) ProcessVote\((M)\)
14. \(\text{if} \ M\ is a timeout message then\) ProcessTM\((M)\)
   ▷ Finally
15. \(\text{procedure TIMER:}\)
16. \(\text{while} \ True\ do:\)
17. \(\text{if} \ \text{timer} \ is \ expired\ then\)
18. \(M.\text{timeout} \leftarrow (vf : currView + 1, vs : \text{vf} + 1)\)
19. \(\text{Broadcast timeout} M_{\text{timeout}} \) to all replicas

1. The main framework of the protocol is provided in Algorithm 1, which is described as message-driven. All messages were generated to drive consensus in a view-by-view loop, and each type of message was given to the corresponding parallel procedure. Further, a replica performed phase in terms of the received message in succession based on its role. The proof of safety and liveness given in Appendix A. The protocol was activated by a new-view message in the initialization step, and then all following runs were based on inner or external messages. A timer mechanism always detected the network fails or message delays to ensure liveness.

   Proposing: There must be two uncertified blocks at any view because of adopting multi-pipeline scheme. A normal commit path implies that the leader proposes and votes in time, and nothing goes bad. However, once the timer detects the failure of the network or leader, the protocol must drop the latest two uncertified blocks and creates a new branch from the latest certified block. In short, one normal and two types of proposing after timeout entails that the multi-pipeline guarantee the safety of the protocol. The predicate for three types of proposing way is presented in Algorithm 2, which guides the protocol to extend and certify block correctly.

   Viewing increasing: At a high level, the synchrony of the network is used to model the transmission latency instead of the ordering of the arrival of messages. Hence, even a message sent first may arrive after messages sent later. The proposing and voting design in multi-pipeline are logically parallel. Consequently, the order in which votes of the previous view and the new proposal are received by a replica is not consistent with the order in which they should be. The view number should be monotonically increasing even if the replica observes a QC that does not belong to the current view. A predicate of consecutive advancing view is presented in Algorithm 2, which prevents block conflicts caused by inconsecutive QC.

   Ordering: The set of committed blocks must be equipped with a total order and certified by three adjacent QC. For instance, the \(b.p\) gives the causal order, indicating that the block \(b\) is directly extend from the block with id \(p\). Further, the block’s view number also can be used to describe the relation. For agreement, every replica should retain the same ordered pairs of the block. Considering the design of multi-pipeline, the unique ordered set is constructed by two parts, the first is that uniqueness of block proposed by the leader, which guarantees the order of the blocks proposed in a monotonically increasing view. The other is that the orderliness of QCs ensures the total order of the ordered block to be certified or committed.

Algorithm 2 utilities for protocol

1. \(\text{function QC}(V)\)
2. \(qc.\sigma \leftarrow tcombine(vote.\rho |\) vote \(\in V)\)
3. \(qc.v \leftarrow vote.\sigma + 1: voting \in V\)
4. \(qc.bid \leftarrow vote.block.id: voting \in V\)
5. \(\text{return} \ qc\)
6. \(\text{function TC}(TM)\)
7. \(tcf.\sigma, tcs.\sigma \leftarrow tcombine(timeout.p |timeout \in TM)\)
8: \[ tcf.v, tcs.v \leftarrow \text{timeout.v}.f, \text{timeout.v}.s+1 \quad \text{timeout} \in TM \]
9: \[ \text{return } \{ tcf, tcs \} \]

10: \textbf{function} \text{ProcessQC(qc)}
11: \quad \textbf{if} currView < qc.v \textbf{then return}
12: \quad currView \leftarrow qc.v + 1
13: \quad \text{secHighQC, highQC} \leftarrow \text{highQC, qc}
14: \quad \text{timer.Reset}()

15: \textbf{function} \text{ProcessTC}(tc, tset)
16: \quad \text{if } (tc = 1) \vee (tset = 1) \textbf{then return}
17: \quad \text{else if } (currView < tc.v) \wedge (tc.v = \min_v \{ v \mid t \in tset \}) \textbf{then}
18: \quad \quad currView \leftarrow tc.v + 1
19: \quad \quad \text{secHighTC, highTC} \leftarrow \text{argmin}_t \{ v \mid t \in tset \}
20: \quad \quad \text{else if } (currView = tc.v) \wedge (tc.v = \max_v \{ v \mid t \in tset \}) \textbf{then}
21: \quad \quad \quad currView \leftarrow tc.v + 1
22: \quad \quad \text{secHighTC, highTC} \leftarrow \bot
23: \quad \quad \text{timer.Reset}()

24: \textbf{function} \text{Verify}(b)
25: \quad \text{if not } \text{teerfy}(b, b, \sigma) \textbf{then return false}
26: \quad \text{if not } \text{teerfy}(b, qc, b, qc, \sigma) \textbf{then return false}
27: \quad \text{return true}

28: \textbf{function} \text{VoteRule}(b)
29: \quad \text{if } b.v \neq currView + 1 \textbf{then return false}
30: \quad \text{if } b.b \neq \bot \textbf{then return true}
31: \quad \text{else if } b.b \leq \min_v \{ v \mid t \in b.b \} \textbf{then return true}
32: \quad \text{else if } b.b \leq \max_v \{ v \mid t \in b.b \} \textbf{then return true}
33: \quad \text{return false}

Algorithm 2 main procedure of MPH

1: \textbf{procedure} \text{ProcessProposal}
2: \quad \text{if not } \text{Verify}(M.block, b) \textbf{then return}
3: \quad \text{ProcessQC}(M.block, b)
4: \quad \text{ProcessTC}(M.block, tc)
5: \quad b' \leftarrow M.block; b'' \leftarrow b.qc.block; b''' \leftarrow b.qc.block;
6: \quad \quad \text{if } (b'.q.b \neq b'.id) \land (b'.q.b = b'.id) \land (b.qc.v > curlockQC.v)
7: \quad \quad \quad \text{then}
8: \quad \quad \quad \text{lastlockQC, curlockQC} \leftarrow \text{curlockQC, b.qc}
9: \quad \quad \text{if } (M.block, tc \neq \bot) \land (M.block, TcSet \neq \bot) \land
10: \quad \quad \quad \text{then}
11: \quad \quad \quad \text{LeaderOf}(M.block, v+1) \text{ then}
12: \quad \quad \quad \text{send newView } M_{\text{newView}}.Vtype \leftarrow \text{TimeoutS}
13: \quad \quad \text{send newView } M_{\text{newView}} \text{ to Proposing procedure}
14: \quad \quad \text{if } (b'.q.b = b'.id) \land (b'.q.b = b'.id) \land
15: \quad \quad \quad \text{then}
16: \quad \quad \quad \text{commit } b''', \text{ reply}
17: \quad \quad \text{if not } \text{VoteRule}(M.block) \textbf{then return}
18: \quad \quad \text{verifiedB} \leftarrow \text{max} \{ \text{verifiedB}, M.block \}
19: \quad \text{Send vote } M_{\text{vote}} \text{ to LeaderOf}(M.block, v+2)

16: \textbf{procedure} \text{PROPOSE(M)}
17: \quad \text{if } i \neq \text{LeaderOf}(currView) \textbf{then return}
18: \quad \text{if } M.Vtype = \text{Normal} \text{ then}
19: \quad \quad b \leftarrow \langle \text{h}(\cdot), currView, \text{h(verifiedB.id), verifiedB.v}, \text{highQC, tset, highTC, tsign(\cdot)} \rangle
20: \quad \quad \text{if } M.Vtype = \text{TimeoutF} \text{ then}
21: \quad \quad \quad \text{b} \leftarrow \langle \text{h}(\cdot), currView, \text{h(highQC.block.id), highQC.block.v}, tset, \text{highQC, secHighTC, secHighTC, tsign(\cdot)} \rangle
22: \quad \quad \quad \text{if } M.Vtype = \text{TimeoutS} \text{ then}
23: \quad \quad \quad \quad \text{b} \leftarrow \langle \text{h}(\cdot), currView, \text{h(verifiedB.id), verifiedB.v},
24: \quad \quad \quad \quad \quad \text{tset, highQC, highTC, secHighTC, highTC, tsign(\cdot)} \rangle
25: \quad \quad \quad \text{M.block} \leftarrow b
26: \quad \quad \quad \text{Broadcast Proposal } M_{\text{proposal}} \text{ to all replicas}
27: \text{end }

26: \textbf{procedure} \text{ProcessVote(M)}
27: \quad V \leftarrow V \cup \text{vote}
28: \quad \text{if } \{ V \} = n - f \textbf{then}
29: \quad \quad \text{qc} \leftarrow QC(V)
30: \quad \quad \text{Wait until currView} = \text{qc}.v + 1 \text{ to consecutive}
31: \quad \text{view transfer}
32: \quad \text{ProcessQC(qc)}
33: \quad \text{M.Vtype} \leftarrow \text{Normal, M.t} \leftarrow \bot
34: \text{end }

34: \textbf{procedure} \text{ProcessTM}(M)
35: \quad TM \leftarrow TM \cup \text{Timeout}
36: \quad \text{if } \{ TM \} = n - f \textbf{then}
37: \quad \quad \text{M.TcSet} \leftarrow TC(TM)
38: \quad \quad \text{ProcessTC(} \text{argmin}_t \{ v \mid t \in M.TcSet \} \text{)}
39: \quad \quad \text{M.Vtype} \leftarrow \text{TimeoutF}
40: \quad \quad \text{Send newView } M_{\text{newView}} \text{ to Proposing procedure}

VII. Evaluation

This section evaluates the performance of the proposed protocol with HotStuff and Streamlet of the same design philosophy (pipelined). Throughput and end-to-end latency were utilized as performance metrics. Throughput is an overall metric that measures the number of transactions that can be committed per second, denoted as TPS. Further, end-to-end latency measures the average time required for a transaction to be submitted to the system until it is committed, including network transmission time, waiting time in transaction buffer, and validation time. As mentioned, the mempool implementation undertakes the transaction body transmission ability for a fairer measurement of the performance. All the compared protocols had the exact shared mempool implementation.

We conducted our experiments on WAN (Alibaba Cloud) and LAN. A testbed on LAN containing 12 local machines, each supported by Intel Xeon Gold 6230 processors (20
A. Best Performance

First, the throughput and latency were measured on a small scale, (4, 10 replicas) replicas and 800 block size (digests). Here, each payload size of the transaction was set to 1024 Bytes, and the mempool’s dissemination batch size was set to 512 KB. The network maintains continuous synchronicity and triggers no view-change.

![Figure 4. Throughput vs. latency with network size 4,10 and batch size 800, on LAN and WAN](image)

Figure 4 depicts the trend of latency with an increase in TPS under the setting of two network conditions and two network sizes. Different transaction arrival rates were set to obtain different TPS when all replicas conducted correct behavior, and the network was synchronous. As can be observed from Figs. 4(a) and 4(b), the setting 4 and 10 replicas were denoted as “-4” and “-10.” Further, the LAN (4(a)) and WAN (4(b)) results showed that regardless of network conditions, the proposed protocol can break through performance bottlenecks and significantly improve TPS with approximately the same latency. As observed from both trends, the latency of the proposed protocol was very close to that of HotStuff, and approximately 60% and 100%, respectively. In addition, benefiting from the concurrent replicas and higher utilization of bandwidth, it can process more transactions in the same number of steps. Owing to its quadratic costs, the throughput of Streamlet is worse than both in this setting.

B. Scalability

The scalability of three protocols with system scales of 4 to 58 was evaluated. A 1024B-size payload, 512KB-size dissemination batch, and a maximum 800-number transaction(digest) were used as the system configuration. Further, the transaction arrival rate was set to be maximum until the system saturated to record the maximum throughput and latency while varying the number of replicas. To capture the error bars of each setting, the standard deviation was calculated for ten runs with the same setting.

![Figure 5. Scalability with batch size 800 on LAN(a) and WAN(b)](image)

Figure 5 shows the throughput and latency over the number of replicas scale-up. As can be observed, the latency of MPH is very close to that of HotStuff, and approximately 60% better throughput than HotStuff at each system scale on LAN 5(a) and WAN 5(b). The throughput of Streamlet drops sharply from the same level as HotStuff at the beginning owing to the quadratic communication cost. When the system scales up to 58 in the LAN, the MPH provides nearly 40k TPS while maintaining an average latency of 2s. The linear communication cost allows HotStuff and MPH to maintain a slow latency growth, whereas, in Streamlet, it increases dramatically. Owing to the transmission delay in the WAN, the latency of all protocols was higher than the LAN environment at each system scale and with a slower increase than the LAN environment except Streamlet. However, the MPH still provided approximately 10k TPS with a maximum latency of 3.5 s, which was better than HotStuff.

C. View-change

Three protocols with 22 replicas were executed with certain being set as faulty (0,1,2 or 3 crashed) at the beginning of...
the experiment on LAN, denoted by “1,2,3.” In addition, payload and dissemination sizes were set to 1024B and 512KB, with maximum 800 block size. Moreover, the timeout was set to 500 ms. Owing to the use of leader rotation in pipeline pattern to provide fairness, this configuration forces the protocol to perform frequent view-change. Therefore, the experiments exhibited the degree to which view-change slowed down the performance of consensus.

Fig. 6. Throughput vs. latency with network size 22 and batch size 800

As evident from Figure 6, MPH outperformed HotStuff and Streamlet at the beginning, and with increase in the number of faulty replicas, the frequent view-change rendered it closer to Hostuff. Because the feature of MPH inherently improves the performance than HotStuff, the throughput loss of MPH for the same number of faulty replicas was lower than other two. Starting from 30k improvement with no faulty replicas to 2k improvement with 3 faulty replicas, the MPH performance gains compared to HotStuff and Streamlet gradually tapered off. The disappearance of multi-pipeline optimization is primarily because of the more complex view-change phase, where more forked blocks than a single pipeline must be processed (at most 4 blocks will be forked in the proposed implementation, twice as many as HotStuff) and only three consecutive views without malicious replicas can effectively drive consensus after the timeout. As for latency, the maximum latency of MPH increased from 200 ms to approximately 2s as more replicas became faulty. When TPS reached its peak, MPH and HotStuff performance were at the same level; however, it was slightly higher than that of Streamlet. The reason why MPH performed better than HotStuff is that more TPS significantly reduced the waiting time in the buffer. It is worse in the case of Streamlet because Streamlet has no view synchronization phase to perform because it follows the longest branch extending rule regardless of the phase.

VIII. Conclusion

This paper proposed MPH, the first protocol that improved performance without the duplication problem caused by parallel proposing. It achieved consensus through the combination of two pipelines in HotStuff into one instance to overcome the performance bottleneck when there are no faults exist. In combination with optimistically proposing, this enabled each leader to produce a block within at most $\Delta$ time after GST, further reducing the waiting time for block generation. The evaluation demonstrated that MPH outperformed other protocols with a proposing-voting pattern in terms of throughput and latency, regardless of the network conditions.

Future research could focus on reducing the dummy blocks when time outs in MPH, less dummy blocks can further improve view-change performance, and, additional HotStuff instances may be incorporated into the multi-pipeline pattern for further optimization.

APPENDIX A
Correctness Proof

Recall that as per the definition of BFT protocol in Section 2.3, it must satisfy two properties, safety, and liveness. If a block is committed by a correct replica, then all other correct replicas should eventually commit the same block in the same view, and subsequently, the committed block must form a linear chain linked by their respective QC. In this Section, the proof of the safety and liveness of MPH is presented.

A. Safety

We start with some definition that we will use:

- $b_j \leftarrow b_j$ indicates that the block $b_j$ extends the block $b_j$
- $b_j \leftarrow qc_{i+1} \leftarrow b_{i+2}$ indicates that the block $b_i$ is certified by the QC of $qc_{i+1}$, which is contained in the block $b_{i+2}$
- $prepareQCView(b_i) := b_{i-4} . v$, such that $b_{i-4} \leftarrow qc_{i-3} \leftarrow b_{i-2} \leftarrow qc_{i-1} \leftarrow b_i$
- $lockedView(r,b_i)$ can return the locked view of replica $r$ after voting for block $b_i$

Lemma 1. Given any valid two QCs, say $qc_1$ and $qc_2$, there must be $qc_1 . v \neq qc_2 . v$.

Proof. We prove this by contradiction, suppose there exist $qc_1 . v = qc_2 . v$. The Section 2 states that a valid threshold signature in QC can be formed only with $n - f = 2f + 1$ partial signatures for it, and each correct replica only votes once. Consider that, the set $R_1$ is the replicas that have voted for block $qc_1 . block$, subject to $|R_1| = 2f + 1$. Likewise, $N_2$ is for $qc_2 . block$, subject to $|N_2| = 2f + 1$. Since the model of system is $n = 2f + 1$, we have $|R_1| + |R_2| - n = 1$ by the quorum intersection, we can conclude that there must be a correct replica who voted twice in the same view. It’s contradict with the assumption, $qc_1 . v = qc_2 . v$ is proved.

Corollary 1. By the Lemma 1, If $qc_1, qc_2$ are two valid QCs, such that $qc_1 . v \neq qc_2 . v$, by the quorum intersection, $R_{qc_1} \cap R_{qc_2} = r$, then there must exist a correct replica $r$ such that $r \in qc_1$ and $r \in qc_2$.

Lemma 2. Consider two certified blocks: $b \leftarrow qc$ and $b' \leftarrow qc'$ under BFT model, if $b . v = b' . v$ then $b = b'$.

Proof. By Lemma 1, we have two valid QCs must be $qc . v \neq qc' . v$, unless $qc = qc'$. Two certified blocks has the same view, by the rule of VoteRule and ProcessQC, there must be $qc = qc'$, then the Lemma 2 is true.

Lemma 3. Assuming that a chain consists of three adjacent certified blocks, which is linked by QC, and start at view $v_0$ and end at view $v_4$. For every certified block $b \leftarrow qc$ such that
\[ b.v > v_0, \text{ then we have that } \text{prepareQCView}(b) > v_0. \]

**Proof.** Let \( b_0 \leftarrow q_{c_0} \leftarrow b_2 \leftarrow q_{c_2} \leftarrow b_4 \leftarrow q_{c_4} \) be that chain starting as view \( b_0.v \) and ending view \( b_4.v \). By Corollary 1, the intersection of QC \( q_{c_i} \) and \( q_{c} \) has at least one correct replica, denote as \( r \). Since \( b.v > v_4 \) and \( \text{VoteRule} \), \( r \) must votes on \( b_4 \) first, then update it's lock on \( b_0 \), that is, \( \text{lockView}(r, b_4) = v_0 \). Because the locked view never decreases, \( r \) votes for the block after \( b_4 \), such as \( b \), the voting rule of \( r \) implies that \( \text{prepareQCView}(b) > v_0 \).

**Lemma 4.** Assuming that a chain with three contiguous view starting with a block \( b_0 \) at view \( v_0 \). For every certified block \( b \leftarrow q_{c} \) such that \( b.v \geq v_0 \), then we have that \( b_0 \leftarrow b. \)

**Proof.** Again, let \( b_0 \leftarrow q_{c_0} \leftarrow b_2 \leftarrow q_{c_2} \leftarrow b_4 \leftarrow q_{c_4} \) be the chain starting with \( b_0 \) and other block's view is contiguous, such that: \( v_0 + 4 = b_0.v + 4 = b_2.v + 2 = b_4.v + 2 \). There are two cases:

1. If \( v_0 \leq b.v \leq v_0 + 4 \), then \( b.v \) can be one of the values: \( v_0, v_0 + 2, v_0 + 4 \). By the Lemma 2, \( b \) is one of blocks: \( b_0, b_2, b_4 \). Hence, \( b_0 \leftarrow b. \)
2. If \( b.v > v_0 + 4 \). By the Lemma 3, we have \( \text{prepareQCView}(b) > v_0 \), this means there exist a block \( b \), which is certified by \( b.qc \), say \( b_i \), such that \( b_i.v \geq v_0 \). Since \( v_0 \leq b.v < b.v \), we could apply same induction hypothesis on \( b_i \) de deduce that \( b_0 \leftarrow b_i. \)

Finally, \( b_0 \leftarrow b \) concludes the proof of \( b_0 \leftarrow b. \)

**Lemma 5.** Consider two conflict blocks: \( b_0, b_0', \) such that \( b_0.qc.p = b_0'.p \) and \( b_0 \neq b_0'. \), only one of them can be committed by a correct replica.

**Proof.** We proof the by contradiction, suppose two blocks can be committed, then each block is extended by two certified block. Let \( b_0 \leftarrow q_{c_0} \leftarrow b_2 \leftarrow q_{c_2} \leftarrow b_4 \leftarrow q_{c_4} \), and \( b_0' \leftarrow q_{c_0}' \leftarrow b_2' \leftarrow q_{c_2}' \leftarrow b_4' \leftarrow q_{c_4}' \) are the commit chain of block \( b_0 \) and \( b_0' \) respectively. By Lemma 2, \( b_0 \neq b_0' \) unless \( b_0.v = b_0'.v \), we assume that \( b_0'.v > b_0.v \). By Lemma 4, we must have \( b_0 \leftarrow b_0' \), this is contradict with \( b_0.qc.p = b_0'.p \) and \( b_0 \neq b_0'. \) Therefore, Lemma 5 is true.

**B. Liveness**

**Lemma 6.** When a replica in view \( v - 1 \) receives a block for view \( v \) from another correct replica, it advances into view \( v \).

**Proof.** If a correct replica create a block for view \( v \), it must have seen votes for \( v - 2 \) or timeout messages for \( v - 1 \), and well formed a QC or TC of view \( v - 1 \). When a correct receives such a block with QC or TC, it will transfer to the next view \( v \).

**Lemma 7.** After GST, the message delays between correct replicas are finite, then all correct replicas keep their view monotonically increasing.

**Proof.** Suppose that all correct replicas start at least view \( v \), and let \( R \) is the set of correct replicas in view \( v \). There are two cases of set \( R \):

1. If all \( 2f + 1 = |R| \) correct replicas time out in view \( v \), then all replicas in \( R \) will receive \( 2f + 1 \) timeout messages to form a TC and enter view \( v + 1 \).
2. Otherwise, at least one correct replica \( r' \), not having sent timeout message for view \( v \), must have observed a QC of view \( v - 1 \) and keep updated its \( q_{c_{high}} \) accordingly. If \( r' \) times out in any view \( v \), then the updated \( q_{c_{high}} \), which is never decreased, must be contained in timeout message, it will trigger \( R \) to enter a view higher than \( v \). Otherwise, \( r' \) must see a QC in all view \( v \). By Lemma 6, the block which contains that QC from correct replica will eventually be delivered to \( R \), triggering it to enter a higher view.

**Lemma 8.** If correct replicas has just entered view \( v \), no QC has yet been generated and none of them has timed out, when correct replica \( r \) receives a block from correct leader in view \( v \), \( r \) will vote for the block.

**Proof.** The predicate in \( \text{VoteRule} \) checks that:

- View numbers are monotonically increasing. By assumption, none of them has timed out means that no TC and QC could have been generated for view \( v \) and \( v - 1 \). Therefore, when predicate of \( \text{VoteRule} \) execute on a block from correct leader is the first largest voting view.
- If TC in block is empty, the block must directly extends the latest verified block in view \( v - 1 \) and in which QC must directly extends must have view \( v - 2 \), all view is consecutive.
- If TC in block is non empty, it is based on \( 2f + 1 \) timeout messages. A correct leader must track the latest QC to have view at least as large as the \( q_{c_{high}} \) of each timeout messages during collecting it. The predicate on QC in block must satisfy with \( q_{c_{lock}}, v - 2 \).

**Lemma 9.** After GST, at most \( 2\Delta \) from the first replica entering view \( v \), all correct replicas receive the block from the correct leader.

**Proof.** When the first correct replica enters view \( v \), if it is leader, all the correct replicas will receive block within \( 2\Delta \); if not, it must have a TC for view \( v - 1 \). Next, it will forward TC to the leader of view \( v \) within \( \Delta \). Upon the leader enters view \( v \), it immediately creates a block and multicasts it to other correct replicas within \( \Delta \).

**Lemma 10.** After GST, for every correct replicas in view \( v \) eventually certify a block \( b \) and commit a block \( b_v \), such that \( b.v > v, b_q \leftarrow q_{c_q} \leftarrow b \leftarrow q_{c} \). \( b \) vote for the block.

**Proof.** At most \( f \) out of \( n \) are faults, such that \( n > 3f + 1 \). Each leader of view designated by round-robin, let \( n = 4, f = 1 \), which is mean that there are three adjacent correct primaries, say \( v', v' + 1, v' + 2, v' > v \). By Lemma 7, all correct replicas keep advancing view. By Lemma 9, at most \( 2\Delta \) from any correct replica enters view \( v', \) the block of \( v' \) will be delivered to all correct replicas. By lemma 8, correct replicas vote for the block from correct leader. Repeatedly deducing the argument by lemma 7.8.9, the block of view \( v', v' + 1, v' + 2, \) say \( b_v, b_v + 1, b_v + 2 \), will append to the chain until next faulty leader comes up. Where at least one \( b_v \) of these new blocks will be certified by QC, which is contained in third block of.
view $b_{v' + 2}$. By the rule of $\text{ProcessQC}$ and $\text{VoteRule}$, when a correct replica successfully perform the vote step on block of view $v' + 2$, the first block $b_v$ among new three adjacent certified blocks linked by QC, will match with commit rule: $b_v \leftarrow \text{certified} \ \text{blocks} \ \leftarrow b_{v' + 1} \leftarrow q_{b_{v' + 1}} \leftarrow \text{commit} \ \text{messages} \ \leftarrow b_{v' + 2}$. Therefore, the block $b_v$ can be certified and $b_v$ can be committed. Each message in view must be delivered within $\Delta$. all correct replica receives block of view $v' + 2$ after at most $4\Delta = 2\Delta + \Delta + \Delta$ time of the first correct replica entering view $v'$.

ACKNOWLEDGMENT

Our deepest gratitude goes to the anonymous reviewers for their careful work and thoughtful suggestions that have helped improve this paper substantially.

REFERENCES

[1] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” ACM Comput. Surv., vol. 22, no. 4, p. 299–319, dec 1990.
[2] L. Lamport, R. Shostak, and M. Pease, The Byzantine Generals Problem. New York, NY, USA: Association for Computing Machinery, 2019, p. 203–226.
[3] R. Pass and E. Shi, “Thunderella: Blockchain with optimistic instant confirmation,” in Advances in Cryptology — EUROCRYPT 2018, J. B. Nielsen and V. Rijmen, Eds. Cham: Springer International Publishing, 2018, pp. 3–33.
[4] V. Shoup, “Practical threshold signatures,” in Advances in Cryptology — EUROCRYPT 2000, B. Preneel, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 207–220.
[5] G. Stathakopoulou, T. David, and M. Vukolic, “Mir-bft: High throughput bft for blockchains,” arXiv preprint arXiv:1906.05532, 2019.
[6] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The honey badger of bft protocols,” in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS ’18. New York, NY, USA: Association for Computing Machinery, 2016, p. 31–42.
[7] J. Garay, A. Kayiayas, and N. Leonardos, “The bitcoin backbone protocol: Analysis and applications,” in Advances in Cryptology - EUROCRYPT 2015, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 281–310.
[8] M. Pease, R. Shostak, and L. Lamport, “Reaching agreement in the presence of faults,” J. ACM, vol. 27, no. 2, p. 229–234, apr 1980.
[9] D. Dolev and H. R. Strong, “Polynomial algorithms for multiple processor agreement,” in Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, ser. STOC ’82. New York, NY, USA: Association for Computing Machinery, 1982, p. 401–407.
[10] M. O. Rabin, “Randomized byzantine generals,” in 24th Annual Symposium on Foundations of Computer Science (sfc83), 1983, pp. 403–409.
[11] J. Katz and C.-Y. Koo, “On expected constant-round protocols for byzantine agreement,” in Advances in Cryptology - CRYPTO 2006, C. Dwork, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 445–462.
[12] I. Abraham, S. Devadas, D. Dolev, K. Nayaq, and L. Ren, “Synchronous byzantine agreement with expected $o(1)$ rounds, expected $o(n^2)$ communication, and optimal resilience,” in Financial Cryptography and Data Security, I. Goldberg and T. Moore, Eds. Cham: Springer International Publishing, 2019, pp. 320–334.
[13] I. Abraham, D. Malkhi, K. Nayaq, L. Ren, and M. Yin, “Sync hoststuff: Simple and practical synchronous state machine replication,” in 2020 IEEE Symposium on Security and Privacy (SP), 2020, p. 106–118.
[14] I. Abraham, K. Nayaq, and N. Shreshtha, “Optimal good-case latency for rotating leader synchronous bft,” Cryptology ePrint Archive, Report 2021/1138, 2021, https://ia.cr/2021/1138.
[15] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” J. ACM, vol. 35, no. 2, p. 288–323, apr 1988.
[16] M. Castro and B. Liskov, “Practical byzantine fault tolerance,” in Proceedings of the Third Symposium on Operating Systems Design and Implementation, ser. OSDI ’99. USA: USENIX Association, 1999, p. 173–186.
[17] J. Kwon, “Tendermint: Consensus without mining,” Draft v 0.6, fall, vol. 1, no. 11, 2014.
[18] R. Kodla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: Speculative byzantine fault tolerance,” SIGOPS Oper. Syst. Rev., vol. 41, no. 6, p. 45–58, oct 2007.
[19] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J.-P. Martin, “Revisiting fast practical byzantine fault tolerance,” arXiv preprint arXiv:1712.01367, 2017.
[20] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, “SbfT: A scalable and decentralized trust infrastructure,” in 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019, pp. 568–580.
[21] M. Yat, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “Hotstuff: Bft consensus with linearity and responsiveness,” in Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, ser. PODC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–356.
[22] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sommiono, “State machine replication in the libra blockchain,” The Libra Lekn, Tech. Rep., 2019.
[23] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai, “Fast-hoststuff: A fast and robust bft protocol for blockchains,” 2021.
[24] B. Y. Chan and E. Shi, “Streamlet: Textbook streamlined blockchains,” in Proceedings of the 2nd ACM Conference on Advances in Financial Technologies, ser. AFT ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–11.
[25] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” J. ACM, vol. 32, no. 2, p. 374–382, apr 1985.
[26] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, “Secure and efficient asynchronous broadcast protocols,” in Advances in Cryptology — CRYPTO 2001, J. Kilian, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 524–541.
[27] I. Abraham, D. Malkhi, and A. Spiegelman, “Asymptotically optimal validated asynchronous byzantine agreement,” in Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, ser. PODC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 337–346.
[28] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, Dumbo: Faster Asynchronous BFT Protocols. New York, NY, USA: Association for Computing Machinery, 2020, p. 803–818.
[29] I. Keidar, E. Kokorits-Kogias, O. Naor, and A. Spiegelman, “All you need is dag,” in Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing, ser. PODC’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 165–175.
[30] D. Boneh, B. Lynn, and H. Shacham, “Short signatures from the weil pairing,” in Advances in Cryptology — ASIACRYPT 2001, C. Boyd, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 514–532.
[31] P. Rogaway and T. Shrimpton, “Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance,” in Fast Software Encryption, B. Roy and W. Meier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 371–388.
[32] F. Gai, A. Farahbakhsh, J. Niu, C. Feng, I. Beschastnikh, and H. Duan, “Dissecting the performance of chained-bft,” in 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), 2021, pp. 595–606.
[33] A. Ailijiang, A. Chazarpo, and M. Demirbas, “Dissecting the performance of strongly-consistent replication protocols,” in Proceedings of the 2019 International Conference on Management of Data, ser. SIGMOD ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1696–1710.
[34] G. Danezis, L. Kokorits-Kogias, A. Sonnino, and A. Spiegelman, “Narwhal and tusks: A dag-based mempool and efficient bft consensus,” in Proceedings of the Seventeenth European Conference on Computer Systems, ser. EuroSys ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 34–50.