Fast-HotStuff: A Fast and Robust BFT Protocol for Blockchains

Mohammad M. Jalalzai*†, Jianyu Niu*†, Chen Feng*† and Fangyu Gai*†
*School of Engineering, University of British Columbia (Okanagan Campus)
†Blockchain@UBC, The University of British Columbia, Vancouver, BC, Canada
{m.jalalzai, jianyu.niu, chen.feng, fangyu.gai}@ubc.ca

***Abstract*—The HotStuff protocol is a breakthrough in Byzantine Fault Tolerant (BFT) consensus that enjoys both responsiveness and linear view change. It creatively adds an additional round to classic BFT protocols (like PBFT) using two rounds. This brings us to an interesting question: Is this additional round really necessary in practice? In this paper, we answer this question by designing a new two-round BFT protocol called Fast-HotStuff, which enjoys responsiveness and efficient view change that is comparable to linear view change in terms of performance. Compared to (three-round) HotStuff, Fast-HotStuff has lower latency and is more robust against performance attacks that HotStuff is susceptible to.**

***Index Terms*—Blockchain, BFT, Consensus, Latency, Performance, Security.**

I. Introduction

Byzantine fault-tolerant (BFT) consensus has received considerable attention in the last decade due to its promising application in blockchains. Several state-of-the-art BFT protocols including Tendermint [1], Pala [2], Casper FFG [3], and HotStuff [4] have been proposed. These protocols not only support linear message complexity by using advanced cryptography like aggregated or threshold signatures, but also enable frequent leader rotation by adopting the chain structure (which is popular in blockchains). What is more, these protocols can be pipelined, which can further improve their performances, and meanwhile, make them much simpler to implement.

The HotStuff protocol is the first to achieve both linear view change[1] and responsiveness, solving a decades-long open problem in BFT consensus. Linear view change enables fast leader rotation while responsiveness drives the protocol to consensus at the speed of actual network delay. Both are desirable properties in the blockchain space. To achieve both properties, HotStuff uses threshold signatures and creatively adopts a three-chain commit rule (which takes three uninterrupted rounds of communication among replicas to commit a block). This is in contrast with the two-chain commit rule commonly used in most other BFT protocols [1]–[3], [5], [6]. In BFT protocols, a decision is made after going through several phases and each phase usually takes one round of communication before moving to the next. Furthermore, HotStuff introduces a chained structure borrowed from blockchain to pipeline all the phases into a unifying propose-vote pattern which significantly simplifies the protocol. Perhaps for this reason, pipelined HotStuff (also called chained HotStuff) has been adopted by Facebook's DiemBFT [7] (previously known as LibraBFT), Flow platform [8], as well as Cypherium Blockchain [9].

To make these achievements (linear view change and responsiveness) HotStuff has made trade-offs. First, it adds a round of consensus to the classic two-round BFT consensus. Secondly, to achieve higher throughput a rotating primary in pipelined HotStuff proposes a block without waiting for the previously proposed block to get committed (proposals are pipelined). This results in the formation of forks which can be exploited by Byzantine primaries to overwrite blocks from honest primaries before they are committed. Hence, forking in HotStuff results in lower throughput and higher latency. It should be noted that classic BFT protocols [5], [10], [11] do not allow fork formation (later we will discuss how forks are formed in HotStuff).

The forking attack in pipelined HotStuff is made possible because a malicious primary can use an old QC[2] instead of the latest one to construct a new proposal. This is valid if the QC is no more than two views older than the latest one according to HotStuff's voting rule. As such, the previous blocks that have not been committed will be forked. In the pipelined HotStuff a child block is proposed after votes for its parent block are collected by the primary, without waiting for the parent block to get committed. This is in contrast with the classic BFT protocols, where the primary does wait for the parent block to get committed before proposing the child block. As a result, this optimization in HotStuff enables the formation of forks (of uncommitted blocks) in pipelined HotStuff, which in turn allows a malicious primary block to override blocks from honest primaries (more details about forking attack is given in Section III).

To address above mentioned problems we designed Fast-HotStuff. Fast-HotStuff's design is shaped by four requirements mentioned below:

- **R1: Low Latency.** Our goal is to reduce the three-round communication latency to two rounds. In this way, blocks can be committed with around 30% less delay.
- **R2: Responsiveness.**: The protocol should operate at the speed of wire rather than waiting for the maximum

[1]In HotStuff [4], the linear view change has been defined as the number of signatures/authenticators sent over the wire during consensus.

[2]QC stands for quorum certificate that contains the parent block hash and $n-f$ votes from replicas for the parent block. Hence in HotStuff, a block uses the QC to point to its parent.

network latency to commit a block. A two-round (two-chain) variant of HotStuff [4], Tendermint [1] and Casper [3] have been built on synchronous core and has to wait for the maximum network delay (lacks responsiveness). But these two-round protocols do not suffer from high view change complexity.

- **R3: Scalability.** Responsive two-round BFT protocols suffer from expensive view change algorithms. During view change, a newly selected primary collects information from replicas in the form of QC to propose the next block/request. In PBFT [5] $O(n^3)$ authenticators/signatures are exchanged and processed during view change. More recent protocols like SBFT [11], Zyzzyva [12] and BFT-Smart [13] have reduced the number of signatures transmitted and processed within the network to $O(n^2)$. Therefore for a large n (network size), the view change cost is high, causing an enormous delay. This becomes an even bigger problem when using a rotating primary protocol where the primary is replaced after each block proposal. Therefore, scalability can be achieved through efficient view change. In HotStuff only one QC[3] is sent and verified during consensus. This is why it achieves linear view change.
- **R4: Robustness Against Forking Attack.** As discussed, forking attacks significantly, reduces the protocol performance in terms of latency and throughput. Therefore, the newly designed protocol should be robust against forking attack.

Fast-HotStuff is designed to fulfill the above-mentioned requirements. The basic idea behind our Fast-HotStuff is simple. Unlike HotStuff, a primary in Fast-HotStuff has to provide a *proof* that it has included the latest QC in the proposed block. The QC in a block points to its parent block. Therefore, by providing proof of the latest/highest QC, a Byzantine primary cannot perform a forking attack (**R4**). It is because now the Byzantine primary cannot use an older QC due to the requirement of the proof of the latest QC included in the block. Moreover, the presence of the proof for the latest QC helps a replica to achieve consensus within two rounds. It provides the guarantee that if a replica committed a block B at the end of the second round, eventually all other replicas will commit the same block at the same height (**R1**). Therefore, a replica does not need to wait for the maximum network delay to make sure other replicas have also committed the block B (**R2**). To achieve the same guarantee HotStuff protocol has to wait for three rounds (see Section III).

In Fast-HotStuff in the absence of primary failure, during normal rotation or happy path[4] of the primary (view change) no overhead is needed. This means when no primary failure occurs then the primary rotation or the view change requires only one QC to send and verified. Therefore, the happy-path in Fast-HotStuff is linear. Unfortunately, we are not able to avoid the transfer of quadratic view change messages over the wire during view change due to primary failure (also called unhappy path). But we were able to reduce the number of signatures to be processed (verified) by each replica by the factor of $O(n)$. The number of authenticators (aggregated signatures) to be verified in Fast-HotStuff during unhappy path is reduced to only two. This is in contrast with other two-chain responsive BFT protocols [**?**], [11]–[14] where $O(n)$ authenticators need to be verified by each replica. Therefore reducing the number of signatures to be verified significantly improves performance. The tradeoff for improvements achieved in Fast-HotStuff is a small overhead each time a primary fails (unhappy path). For example, this overhead is at most $\approx 1.4\%$ of the block size (1MB) and $\approx 0.7\%$ of $2MB$ block for the network size of 100 replicas. We also show in Sections IV and V that this overhead does not have a significant impact on performance. Overall, during the unhappy-path, the view change is optimized by reducing the number of authenticators to be verified by each replica. Whereas during happy path the protocol enjoys linear view change (**R3**).

The rest of the paper is organized as follows. In Section II, we describe the system model. In Section III, we present a brief overview of classic HotStuff protocol. Section IV provides algorithms for basic Fast-HotStuff and pipelined Fast-HotStuff protocols as well as their safety and liveness proofs. This section also discusses how pipelined Fast-HotStuff is robust to forking attacks. Evaluation and related work are presented in Sections V and VI, respectively. The paper is concluded in Section VII.

II. System Model and Preliminaries

A. System Model

We consider a system with $n = 3f + 1$ parties (also called replicas) denoted by the set N such that the system can tolerate at most f Byzantine replicas. Byzantine replicas may behave in an arbitrary manner, whereas correct (or honest) replicas follow the protocol which results in the execution of identical commands at the same order. We assume a partial synchronous model presented in [15], where the network is synchronous with a known bound on message transmission delay denoted by Δ. This known bound on message transmission holds after an unknown asynchronous period called *Global Stabilization Time* (GST). All exchanged messages are signed. Adversaries are computationally bound and cannot forge signatures or message digests (hashes) but with negligible probability.

B. Preliminaries

Signature Aggregation. Fast-HotStuff uses signature aggregation [16]–[18] to obtain a single collective signature instead of appending all replica signatures when a primary fails. As the primary p receives the message M_i with their respective signatures $\sigma_i \leftarrow sign_i(M_i)$ from each replica i, the primary then uses these received signatures to generate an aggregated

[3]HotStuff uses threshold signature and Fast-HotStuff uses aggregated signature scheme. But in practice, we show that both signature schemes have comparable performance.

[4]HotStuff and Fast-HotStuff both use a rotating primary mechanism, therefore there are mainly two types of view change 1) Happy path, in which a block is successfully proposed and $n - f$ votes are collected for it and 2) the primary failed to propose block, other replicas timeout and move to the next view (primary).

signature $\sigma \leftarrow AggSign(\{M_i, \sigma_i\}_{i\in N})$. The aggregated signature can be verified by replicas given the messages $M_1, M_2, \ldots, M_y$ where $2f+1 \leq y \leq n$, the aggregated signature σ, and public keys $PK_1, PK_2, \ldots, PK_y$. To authenticate message senders as done in previous BFT-based protocols [5], [10], [19] each replica i keeps the public keys of other replicas in the network.

Quorum Certificate (QC) and Aggregated QC. A block's B quorum certificate (QC) is proof that more than $2n/3$ nodes (out of n) have voted for this block. A QC comprises an aggregated signature or threshold (from $n-f$ signatures from distinct replicas) built by signing block hash in a specific view. A block is certified when its QC is received and certified blocks' freshness is ranked by their view numbers. In particular, we refer to a certified block with the highest view number that a node knows as the *latest/highest* certified block. The latest *QC* a node knows is called *highQC*. A node keeps track of *QC*s for each block and keeps updating the *highQC* as new blocks get certified. It should be noted that unlike classic BFT in HotStuff and Fast-HotStuff the *QC* is also used to point to the parent block. An aggregated *QC* or *AggQC* is simply a vector built from a concatenation of $n-f$ or $2f+1$ *QCs*.

View and View Number. During each view, a dedicated primary is responsible for proposing a block. Each view is identified by a monotonically increasing number called view number. Each replica uses a deterministic function to translate the view number to a replica *ID*, which will act as primary for that view. Therefore, the primary/leader of each view is known to all replicas. Similar to the HotStuff, in Fast-HotStuff primaries are selected in a round-robin manner.

Block and Block Tree. Clients send transactions to primary, who then batch transactions into blocks. Block data structure also has a field that it uses to point to its parent. In the case of HotStuff and Fast-HotStuff, a *QC* (which is built from $n-f$ votes) is used as a pointer to the parent block. Every block except the genesis block must specify its parent block and include a QC for the parent block. In this way, blocks are chained together. As there may be forks, each replica maintains a block tree (referred to as *blockTree*) of received blocks. Two blocks B_{v+1} and B_{v+2} are conflicting if they have a common predecessor block (B_v in this case). The parent block B_v is the vertex of the fork as shown in Figure 1.

Chain and Direct Chain. If a block B is being added over the top of a block B' (such that $B = B'.parent$), then these two blocks make one-chain. If another block $B*$ is added over the top of the block B, then B, B', and $B*$ make two-chain and so on. There are two ways that the chain or tree of blocks grows. First, the chain grows in a continuous manner where the chain is made between two consecutive blocks. For example for two blocks B and B' we have $B.curView = B'.curView + 1$ and $B = B'.parent$. Thus a direct chain exists between block B and B'. But there is also the possibility that one or more views between two views fail to generate block due to primary being Byzantine or network failure. In that case $B.curView = B'.curView + k$ where $k > 1$ and $B = B'.parent$ and direct chain does not exist between B and B'.

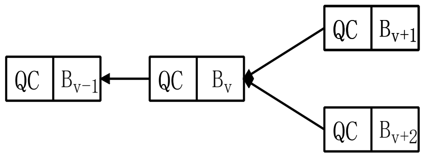


Fig. 1. **A simple case of conflicting blocks**. Block B_v is fork vertex, and blocks B_{v+1} and B_{v+2} are two conflicting blocks.

III. HotStuff in a nutshell

In this section, we provide a brief description of the three-chain HotStuff protocol. There are two variants of the three-chain HotStuff protocol, the basic and pipelined HotStuff. Since pipelined HotStuff is mainly used due to its higher throughput, here, we will focus on pipelined HotStuff.

A. Pipelined HotStuff

We describe the pipelined HotStuff operation beginning in the view v. At the beginning of the view v a dedicated primary is selected by all replicas operating in the view number v. The primary proposes a block B_v (broadcast to all replicas) that extends the block certified by the *highQC* it has seen. Upon receipt of the first proposed block from the primary, each replica sends its vote (a signature on the block) to the primary of view v. Upon receipt of $n-f$ votes, the primary of the view v builds a *QC* and forwards the *QC* to the primary of the view $v+1$. The *QC* of the view v is the *highQC* that the primary of the view $v+1$ is holding. Therefore, the primary in the view $v+1$ will propose its block (B_{v+1}) with the *QC* from the view v as shown in the Figure 2.

Every replica has to keep track of two local parameters in HotStuff 1) $last\_voted\_view$, the latest view when the replica has voted and 2) $last\_locked\_view$, the view of the grandparent block of the $last\_voted\_view$. As it can be seen in the figure 2, when a replica votes for the block B_{v+2} during view $v+2$ its $last\_voted\_view$ will be $v+2$ and it locks the grandparent block of the block B_{v+2}. Hence, $last\_locked\_view$ will be the view v. In order to vote each replica makes sure that the received block satisfies the voting conditions. Voting conditions include 1) the proposed block extends the block at the $last\_locked\_view$ or 2) the view number of the received block's parent is greater than the $last\_locked\_view$. If any of the voting conditions are satisfied then the replica will vote and update its $last\_voted\_view$ and the $last\_locked\_view$ to the new blocks grandparent if the latter has a higher view number than the current $last\_locked\_view$. For example, when the replica receives the block for view $v+3$ (B_{v+3}) it then makes sure voting conditions are met. As it can be seen, the block B_{v+3} extends the $last\_locked\_view$ (view v), satisfying the first condition (though second condition is also satisfied as view of the parent of B_{v+3} is greater than v or $v+2 > v$). Therefore the replica will vote for B_{v+3}. The replica then increments its $last\_voted\_view$ to $v+3$, its $last\_locked\_view$ to $v+1$. Then the replica checks its *block tree* to see if there is any block that needs to be committed. If there are three blocks added over the top of each other during consecutive (uninterrupted) views

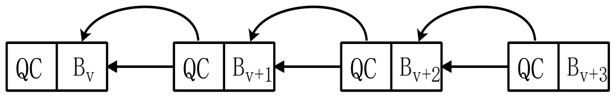


Fig. 2. **The chain structure of pipelined HotStuff.** Curved arrows denote the Quorum certificate references.

(B_v, B_{v+1}, B_{v+2}) and extended by at least one another block (in this case B_{v+3}), then the replica will commit the block B_v and all its predecessor blocks. The first three blocks added to the chain during consecutive uninterrupted views make two-direct chain. For the first block in the two-direct chain to get committed, an additional block (may not form a direct chain) is need to extend it and make it a three-chain.

Overall, the HotStuff primaries propose blocks, and replicas vote on the proposed blocks. Replicas commit a block if there is a two-direct chain extended by another block making 3-chain. Replicas movie to the next view either when they receive a valid block with a *QC* or when the primary fails to propose the block, and replicas timeout (wait for a proposal until timeout period). In either case, the pacemaker module provides view synchronization.

Three-round vote collection in HotStuff. As we discussed previously, the primary in the view $v+1$ collects votes for the view $v+1$, builds a *QC*, and will forward it to the primary of the view $v+2$. This *QC* will be the *highQC* (*QC* with the highest view) for primary in view $v+2$. The primary of the view $V+2$ will then include this *QC* into its block and will propose it during the view $V+2$. As it can be seen, replicas do not broadcast their votes as the classic PBFT protocol [5]. The distribution of *QC* through the primary (within a block), gives rise to the possibility that if the primary fails, then a subset of replicas may receive the *QC*.

Now we consider if HotStuff changes its commit rule to a two-chain while not waiting for a three-chain chain, then it may result in an infinite non-deciding scenario. If the primary of view $v+2$ fails after sending its block B_{v+2} along with the *QC* for the view $v+1$ to a single replica i, the replica i will lock the block B_{v+1}(in three-chain rule replica i will lock the grandparent of the received block or B_v). Since other replicas have not received block B_{v+2}, they timeout and move to the next view while having their lock on view v (B_v).

The next primary in the $v+3$ will receive $2f+1$ view change messages from replicas (excluding i). Since other replicas have *QC* for the view v as the highest *QC* (*highQC*) therefore the primary $v+3$ proposes a block using the *QC* for the view v. Since replica i has locked the view $v+1$, it will reject the competitive proposal. While other $2f$ replicas vote as each of them has locked on the view v. Since the primary, only collects $2f$ votes it cannot prepare a *QC*. Therefore, replicas in the view $v+3$ timeout and move to the next view. Just during this time a Byzantine replica sends its vote to the primary and the primary builds a *QC*, sending it to only one honest replica before it fails. Therefore, the other honest replica gets locked into the view $v+3$. Since honest replicas are getting locked, other proposals will not be able to collect enough votes causing an indefinite non-deciding scenario. Therefore, HotStuff has to lock after two rounds of communication and commit after three rounds.

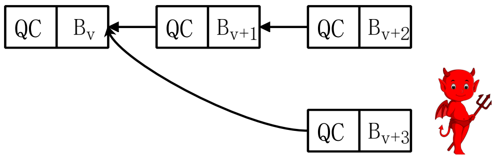


Fig. 3. **Forking attack by the primary of the view v+3.**

Forking Attack. A Byzantine primary in pipelined HotStuff can deliberately generate forks to override the blocks from the honest primary. As it can be seen in Figure 3, blocks B_{v+1} and B_{v+2} are honest blocks and the Byzantine primary will propose block B_{v+3} using the *QC* for the block B_v. Other replicas will vote for B_{v+3} as it satisfies the voting rule (B_{v+3} parent's view is at least equal to the B_v view (v) or B_{v+3} extends B_v). Since B_{v+3} has higher view than B_{v+2}, next primary will extend B_{v+3}. As a result, the resources (computing, bandwidth, etc.) spend on these blocks are wasted. Transactions of reverted blocks will have to be re-proposed by another primary. Hence, these transactions encounter additional latency. Moreover, since forking overrides the honest blocks it also breaks the direct chain relation among blocks (refer to Section II). This results in delaying the block commitment as the commit rule require that for a block to get commit, at least the first two blocks over the top of it should make a 2-direct chain (two blocks from consecutive views). Forking attack makes it difficult to design an incentive mechanism like PoS over the top of HotStuff as blocks from honest primaries will not be able to get committed and hence honest primaries will less likely to earn rewards[5].

IV. Fast-HotStuff

Fast-HotStuff operates in a series of views with monotonically increasing view numbers. Each view number is mapped to a unique dedicated primary known to all replicas. The basic idea of Fast-HotStuff is simple. The primary has to convince voting replicas that its proposed block is extending the block pointed by the *highQC*. Once replicas are convinced that the proposed *QC* is the *highQC*, it checks if two-direct chain is formed over the top of the parent of the block pointed by the *highQC*. Then a replica can safely commit the parent of the block pointed by the *highQC*.

Unlike HotStuff (where a block is committed in three rounds), in the Fast-HotStuff protocol, a replica i can optimistically commit/execute a block during at the end of the second round while guaranteeing that all other correct replicas will also commit the same block in the same sequence eventually. This guarantee is valid when either one of the two conditions is met: **1) the block proposed by the primary is built by using a** ***QC*** **that is the** ***highQC*** **held by the majority of the honest replicas, or 2) by a** ***QC*** **higher than the** ***highQC*** **being held by a majority of replicas.** Therefore, the primary has to incorporate proof of the *highQC* in every block it proposes, which can be verified by every replica.

Interestingly, the inclusion of the proof of the *highQC* by the primary in Fast-HotStuff also enables us to design a responsive

[5]More details about this is part of our future work

two-chain consensus protocol. Indeed, the presence of the proof for the *highQC* guarantees that a replica can safely commit a block after two-chain without waiting for the maximum network delay as done in two-chain HotStuff [4], Tendermint [1], and Casper [3]. Therefore, Fast-HotStuff achieves responsiveness only with a two-chain structure (which means two rounds of communication) in comparison to the three-chain structure in HotStuff (three rounds of communication).

Algorithm 1: Utilities for replica *i*

```
Func CreatePrepareMsg(type, aggQC, qc, cmd):
    b.type ← type
    b.AggQC ← aggQC
    b.QC ← qc
    b.cmd ← cmd
    return b
End Function
Func GenerateQC(V):
    qc.type ← m.type : $m \in V$
    qc.viewNumber ← m.viewNumber : $m \in V$
    qc.block ← m.block : $m \in V$
    qc.sig ← AggSign( qc.type, qc.viewNumber, qc.block,i,
      {m.Sig |$m \in V$})
    return qc
End Function
Func CreateAggQC($\eta_{Set}$):
    aggQC.QCset ← extract QCs from $\eta_{Set}$
    aggQC.sig ← AggSign( curView,
      $\{qc.block | qc.block \in aggQC.QCset\}$, $\{i | i \in N\}$,
      $\{m.Sig | m \in \eta_{Set}\}$)
    return aggQC
End Function
Func BasicSafeProposal(b,qc):
    return $b$ $extends$ $from$ $highQC.block$
End Function
Func PipelinedSafeBlock(b,qc, aggQC):
    if $QC$ then
        return $b.viewNumber \geq curView \wedge b.viewNumber ==$
          $qc.viewNumber + 1$
    end
    if $AggQC$ then
        $highQC$ ← extract highQc from $AggregatedQC$
        return $b$ $extends$ $from$ $highQC.block$
    end
End Function
```

Since HotStuff has two three-chain variant (the basic and the pipelined HotStuff), therefore we present two-chain basic Fast-HotStuff as well as pipelined Fast-HotStuff protocols. First, we present the basic optimised two-chain Fast-HotStuff and then extend it to an optimized two-chain pipelined Fast-HotStuff protocol. As mentioned earlier the basic Fast-HotStuff is two-chain protocol and uses the aggregated signature scheme. Moreover, basic Fast-HotStuff needs the proof for the *highQC* in the form of $n-f$ *QC*s attached in the proposed block by the primary in order to be able to guarantee safety and liveness as a two-chain protocol. In the case of blockchains since block size is usually large (multiples of Megabytes), this overhead has little effect on performance metrics like throughput and latency. We also show that the protocol requires only two *QC*s to be verified instead of $n-f$ *QC*s. Then we further optimized pipelined Fast-HotStuff by introducing proposal pipelining. Moreover, for pipelined Fast-HotStuff, the block include a single *QC* during happy path and similar to the basic Fast-HotStuff only two *QC*s need to be verified in case of the primary failure.

A. Basic Fast-HotStuff

The algorithm for two-chain Responsive basic HotStuff is given in Algorithm 2. Below we describe how the basic Fast-HotStuff algorithm operates. Fast-HotStuff operates in three phases PREPARE, PRECOMMIT and COMMIT. The function of each phase is described below.

PREPARE phase. Initially the primary replica waits for new-view messages $\eta = \langle "NEWVIEW", curView, prepareQC, i\rangle$ from $n-f$ replicas. The NEWVIEW message contains four field indicating message type (NEWVIEW), current view (*curView*), the latest *PrepareQC* known to replica *i*, and replica id *i*. The *NEWVIEW* message is signed by each replica over fields $\langle curView, prepareQC.block, i\rangle$. *prepareQC.block* is presented by the hash of the block for which *prepareQC* was built (hash is used to identify the block instead of using the actual block). The primary creates and signs a prepare message $(B = \langle "Prepare", AggQC, commands, curView, h, i\rangle)$ and broadcasts it to all replicas as shown Algorithm 2. We can also use the term block proposal or simply block for *PREPARE* message. *AggQC* is the aggregated *QC* build from valid η messages collected from $n-f$ replicas.

Upon receipt of prepare message B from the primary, a replica i verifies *AggQC*, extracts *highQC* as well from *PREPARE* message and then checks if the proposal is safe. Verification of *AggQC* involves verification of aggregated signatures built from $n-f$ η messages and verification of the latest *PrepareQC*. Signature verification of each *QC* is not necessary, a replica only needs to make sure messages are valid. *BasicSafeProposal* predicate makes sure a replica only accepts a proposal that extends from the *highQC.block*.

If a replica notices that it is missing a block it can download it from other replicas. At the end of PREPARE phase, each replica sends its vote v to the primary. Any vote message v sent by a replica to the primary is signed on tuples $\langle type, viewNumber, block, i\rangle$. Each vote has a type, such that $v.type \in \{PREPARE, PRECOMMIT, COMMIT\}$ (Here again block hash can be used to represent the block to save space and bandwidth).

PRECOMMIT phase. The primary collects PREPARE votes from $n-f$ replicas and builds *PrepareQC*. The primary then broadcasts *PrepareQC* to all replicas in PRECOMMIT message. Upon receipt of a valid PRECOMMIT message, a replica will respond with a PRECOMMIT vote to the primary. Here (in line 25-27 Algorithm 2) replica *i* also checks if it has committed the block for *highQC* called *HighQC.block*. Since the majority of replicas have voted for *HighQC.block*, it is safe to commit it.

COMMIT phase. Similar to PRECOMMIT phase, the primary collects PRECOMMIT votes from $n-f$ replicas and combines them into *PrecommitQC*. As a replica receives and verifies the *PrecommitQC*, it executes the commands. The replica increments *newNumber* and begins the next view.

New-View. Since a replica always receives a message from the primary at a specific viewNumber, therefore, it has to wait for a timeout period during all phases. If NEXTVIEW(viewNumber) utility interrupts waiting, the replica increments viewNumber and starts next view.

Algorithm 2: Basic Fast-HotStuff for replica i

```
foreachin curView ← 1,2,3,...
    ▷ Prepare Phase
    if i is primary then
        wait until $(n-f)$ $\eta$ messages are received:
          $\eta_{Set} \leftarrow \eta_{Set} \cup \eta$
        $aggQC \leftarrow CreateAggQC(\eta_{Set})$
        B ← CreatePrepareMsg(Prepare,aggQC, client's
          command)
        broadcast B
    end
    if i is normal replica then
        wait for prepare B from primary(curView)
        HighQC ← extract highQc from B.AggQC
        if BasicSafeProposal(B, HighQC) then
            Send vote v for prepare message to
              primary(curView)
        end
    end
    ▷ Pre − Commit Phase
    if i is primary then
        wait for $(n-f)$ prepare votes: V ← $V \cup v$
        PrepareQC ← BasicGenerateQC(V)
        broadcast PrepareQC
    end
    if i is normal replica then
        wait for PrepareQC from primary(curView)
        Send Precommit vote v to primary(curView)
        if have not committed HighQC.block then
            commit HighQC.block
        end
    end
    ▷ Commit Phase
    if i is primary then
        wait for $(n-f)$ votes: $V \leftarrow V \cup v$
        PrecommitQC ← GenerateQC(V)
        broadcast PrecommitQC
    end
    if i is normal replica then
        wait for PrecommitQC from primary(curView)
        execute new commands through PrecommitQC.block
        respond to clients
        ▷ New-View
        check always for nextView interrupt then
            goto this line if nextView(curView) is called
              during "wait for" in any phase
            Send $\eta$ to primary($curView+1$)
        end
    end
end
```

B. Correctness Proof for Basic Fast-HotStuff

Correctness of a consensus protocol involves proof of safety and liveness. Safety and liveness are the two important properties of consensus algorithms. In this section, we provide proofs for safety and liveness properties of Fast-HotStuff. We begin with two standard definitions.

Definition 1 (Safety)**.** *A protocol is safe if the following statement holds: if at least one correct replica commits a block at the sequence (blockchain height) s in the presence of f Byzantine replica, then no other block will ever be committed at the sequence s.*

Definition 2 (Liveness)**.** *A protocol is alive if it guarantees progress in the presence of at most f Byzantine replica.*

Next, we introduce several technical lemmas. The first lemma shows that for each view, at most only block can get certified (get $n-f$ votes).

Lemma 1. *If any two valid QCs, qc_1 and qc_2 with same type $qc_1.type = qc_2.type$ and conflicting blocks i.e., $qc_1.block = B$ conflicts with $qc_2.block = B'$, then we have $qc_1.viewNumber \neq qc_2.viewNumber$.*

Proof. We can prove this lemma by contradiction. Furthermore, we assume that $qc_1.viewNumber = qc_2.viewNumber$. Now let's consider, N_1 is a set of replicas that have voted for for block B in $qc_1(|N_1| \geq 2f+1)$. Similarly, N_2 is another set of replicas that have voted for block B' and whose votes are included in qc_2 $(|N_2| \geq 2f+1)$. Since $n = 3f+1$ and $f = \frac{n-1}{3}$, this means there is at least one correct replica j such that $j \in N_1 \cap N_2$ (which means j has voted for both qc_1 and qc_2). But a correct replica only votes once for each phase in each view. Therefore, our assumption is false and hence, $qc_1.viewNumber \neq qc_2.viewNumber$. □

The second lemma proves that if a single replica has committed a block at the view v then all other replicas will commit the same block at the view v.

Lemma 2. *If at least one correct replica has received PrecommitQC for block B, then the PrepareQC for block B will be the highQC (latest QC) for next (child of block B) block B'.*

Proof. The primary begins with a new view $v+1$ as it receives $n-f$ NEWVIEW messages. Here for ease of understanding, we assume $v+1$, basically B' could have been proposed during any view $v' > v$. We show that any combination of $n-f$ NEWVIEW messages have at least one of those NEWVIEW message received by the primary containing *highQC* (*PrepareQC* for block B) for block B.

We know that in previous view v, a set of replicas R_1 of size $|R_1| \geq 2f+1$ have voted for *PrepareQC* (which is built from votes for block B). Similarly, another set of replicas R_2 of size $|R_2| \geq 2f+1$ have sent their NEWVIEW messages to the primary of view $v+1$ after the end of view v . Since the total number of replicas is $n = 3f+1$ and $f = \frac{n-1}{3}$, therefore, $R_1 \cap R_2 = R$, such that, $|R| \geq f+1$, which means that there is at least one correct replica in R_2 that has sent its *PrepareQC* as *highQC* in NEWVIEW message to the primary. Therefore, when primary of view $v+1$ proposes B', the *PrepareQC* for block B will be the *highQC*. In other words B' will point to B as its parent (through *highQC*). □

The third lemma proves that if a replica commits a block then it will not be reverted.

Lemma 3. *A correct replica will not commit two conflicting blocks.*

Proof. From lemma 1 we know that each correct replica votes only once for each view and therefore view number for conflicting blocks B and B' will not be the same. Therefore, we can assume that $B.curView < B'.curView$. Based on lemma 2 we know that if at least one correct replica has received *PrecommitQC* for block B (have committed B), then the *PrepareQC* for block B will be the *highQC* for the next block B' (child of block B). Therefore, any combination of $n-f$ *PrepareQC*s in *AggQC* for B' will include at least one *highQC* such that $highQC.block = B$ or $highQC.block.view = B.view$. But for B' to be conflicting with B it has to point to the parent of B at least. Consequently, first it is not possible for a primary to build a valid PREPARE message B' in which $highQC.block.view < B.view$. Secondly, if a primary tries to propose a block with invalid *AggQC*, it will be rejected by the replica based on Algorithm 2 line 12. □

Lemmas 1, 2 and 3 provide a safety proof for basic Fast-HotStuff consensus protocol. To ensure liveness, Fast-HotStuff has to make sure in each view a replica is selected as a primary and the view number is incremented. Moreover, we should also show that the protocol will eventually add a block to the chain/tree of blocks (a block to the blockchain) or will result in view change in case of failure.

In Fast-HotStuff a new primary is chosen deterministically in a round-robin manner. If a replica times out, it employs exponential-backoff used in PBFT [5] to double its timeout value. This guarantees that eventually during GST there will be an interval T_f when timeout values from all correct replicas intersect and this bounded period is enough to reach a decision during consensus. Below we provide liveness proof for our Fast-HotStuff protocol.

Lemma 4. *After GST, there is a time T_f, when there is an honest primary and all correct/honest replicas are in the same view. As a result, a decision is reached during T_f.*

Proof. Based on lemma 2, at the beginning of a view, the primary will have the latest *PrepareQC* or *highQC* from $n-f$ replicas. As per assumption, all correct replicas are in the same view, therefore the correct primary will propose a PREPARE message with *AggQC* containing the latest *PrepareQC*, which is extracted by each replica. Since all replicas are in the same view until bounded T_f time, therefore all replicas successfully complete PREPARE, PRECOMMIT, and COMMIT phase. □

Efficient View Change (R3) As a replica moves to the next view, it will send its *NEWVIEW* (η) to the next primary. The primary aggregates $n-f$ η messages and their signatures into an *AggQC* and sends back to all replicas. Upon receipt of a block containing an *AggQC*, first the aggregated signature for the $n-f$ (η) messages needs to be verified. The first check (verification of aggregated signature of *AggQC*) verify that the *AggQC* that is built from η messages has $n-f$ valid *QC*s (*QC*s come from $n-f$ distinct replicas). It further guarantees that at least $f+1$ *QC*s out of $n-f$ are from honest replicas. Based on Lemma 2 we know that out of these $f+1$ *QC*s from honest replicas at least one of them is a valid *highQC*.

Therefore, a replica only needs to find a *QC* with highest view among *QC*s in *AggQC* by looping over the view numbers of *QC*s. Next, the replica has to verify the aggregated signature of *highQC* or latest *QC*. As a result, we do not need to verify the remaining $n-f-1$ *QC*s as a replica only needs to verify *highQC* and make sure if the block extends *highQC.block*. This helps to reduce the signature verification cost for *AggQC* ($n-f$ *QC*s) during view change by the factor of $O(n)$ independent of signature scheme used. For example, the the quadratic verification cost of *AggQC* in [17] can be reduced to linear. This is due to the fact that the verification cost of each aggregated signature is linear in [17]. Therefore, each replica has to verify only two *QC*s. Similarly linear cost of *AggQC* in [20] can be reduced to constant for each replica because verification cost of each aggregated signature (in [20]) is constant and each replica has to verify only two *QC*s. This optimization can be used by any two-chain BFT-based consensus protocol during view change where replicas have to receive and process $n-f$ *QC*s.

There is a possibility that the η messages containing *highQC* are invalid. This means η does not meet formatting requirements or its view number is incorrect. In this case, the replica can simply reject the block proposal.

C. Pipelined Fast-Hotstuff

Pipelined Fast-HotStuff has been optimized in different ways in comparison to the basic Fast-HotStuff. First similar to the piplined HotStuff, it pipelines requests and proposes them in each phase to increase the throughput. Secondly, during normal view change when no primary failure occurs the protocol does not require additional overhead. But the proof of the latest/highest *QC* in the block in pipelined Fast-HotStuff carries a small overhead (*AggQC*) in the block during view v if the primary in view $v-1$ fails.

Blocks can be added into the chain during happy path with no failure or the primary fails and the next primary will have to add its block into the chain. Unlike HotStuff, pipelined Fast-HotStuff addresses these two cases differently. Indeed, there are two ways a primary can convince replicas that the proposed block extends the latest *QC* (*highQC*). In case of contiguous chain growth, a primary can only propose a block during view v if it is able to build a *QC* from $n-f$ votes received during the view $v-1$. Therefore in a contiguous case, each replica signs the vote , the primary will build a *QC* from $n-f$ received votes and include it in the next block proposal. Therefore, the block will contain only the *QC* for view $v-1$. As a result, for the block during view v, the *QC* generated from votes in $v-1$ is the proof of *highQC*. It should be noted that in pipelined Fast-HotStuff, a replica commits a block if two direct chain is formed over the top of it. In other words, a block is committed if two blocks from consecutive views are added over the top of it. Similarly, if a primary during view v did not receive $n-f$ votes from view $v-1$ (this means the primary during view $v-1$ has failed), then it can only propose a block if it has received $n-f$ η (NEWVIEW) messages from distinct replicas for view v. In this case, the primary during view v has to propose a block with aggregated *QC* or *AggQC* from $n-f$ replicas (**R1, R2**).

The signatures on η are aggregated to generate a single aggregated signature in *AggQC*. Therefore, there are basically two types of blocks that can be proposed by a primary: a block with *QC* if the primary is able to build a *QC* from previous view or a block with *AggQC* if the previous primary has failed. It should be noted that to verify *AggQC* a replica only needs to verify the aggregated signature of *AggQC* and the *highQC* in the *AggQC* as described previously. The *highQC* can simply be found by looping over view numbers of each *QC* and choose the highest one (**R3**).

The chain structure in pipelined Fast-HotStuff is shown in Figure 4. Here, we can see that upon receipt of block B_{v+2} (during the view $v+2$), a two-chain with direct chain is completed for the block B_v. Now a replica can simply execute the block B_v. Since there is no primary failure for views v through view $v+2$, only *highQC* is included in the block proposal. Similarly, the primary for the view $v+3$ has failed, therefore, the primary for the view $v+4$, has to propose a block with *AggQC* (small overhead). Upon receipt of B_{v+4}, the *highQC* (*highQC*) can be extracted from *AggQC*. In this case, the *QC* for the block B_{v+2} has been selected as the *highQC*.

As it can be seen the algorithm mainly has two components: the part executed by the primary and the part run by replicas. The *primary* either receives votes that it will aggregate into a *QC* or NEWVIEW messages containing *QC* from $n-f$ replicas that the primary will aggregate into *AggQC* (Algorithm 3 lines 23-25 and 2-9). The primary then builds a proposal in the form of PREPARE message also called a block. PREPARE also contains *QC* or *AggQC* depending on if it has received $n-f$ votes or NEWVIEW messages (Algorithm 3 lines 2-11). The primary then proposes the block to replicas ((Algorithm 3 line 11)).

Upon receipt of block B (containing a *QC*) through a proposal, each replica can check the condition $B.viewNumber == B.QC.viewNumber + 1$ and accept the proposal if the condition is met. On other hand, if the block contains a valid *AggQC*, then each replica extracts the *highQC* from *AggQC* and checks if the block extends the *highQC.block*. These checks are performed through *PipelinedSafeBlock* predicate (Algorithm 1). If the check was successful, each replica sends back a vote to the next primary (Algorithm 3 lines 13-16). After that, each replica commits the grandparent of the received block if direct chain is formed between the received block and its parent as well as parent of the received block and its grand parent (Algorithm 3, lines

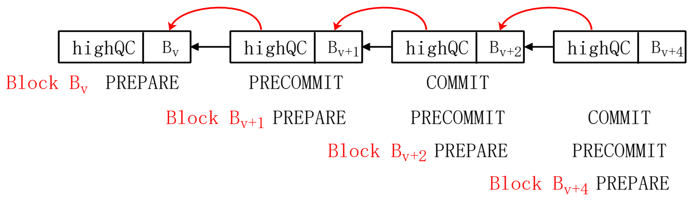


Fig. 4. **Pipeined/Chained Fast-HotStuff where a QC can serve in different phases simultaneously. Note that the primary for view $v+3$ has failed.**

17-21). In other words, a block is committed if two blocks are built over top of it in consecutive views (two-chain is complete) without waiting for the maximum network delay (**R1, R2**).

Resilience against Forking Attack (R4). Unlike pipelined HotStuff, pipelined Fast-HotStuff is robust to forking attacks. In Fast-HotStuff the primary has to provide the proof of the latest *QC* included in the block. This proof can be provided in two ways. First, if there is no primary failure, then for the proposed block $B*$ and the *QC* it contains (*qc*), we have $B.view = qc.view + 1$. Secondly, if there is a primary failure in the previous view than the primary has to include the *AggQC* ($n-f$ *QC*s). This guarantees at least one of the *QC*s in the *AggQC* is the latest *QC* held by the majority of replicas or a *QC* higher than the latest *QC* being held by the majority of replicas. The inclusion of proof within block prevents the primary from using an old *QC* to generate forks. If a primary does not provide appropriate proof, its proposal can be rejected. Furthermore, such a proposal can be used as a proof to blacklist the primary.

D. Correctness Proof for Pipelined Fast-HotStuff

We can establish the safety and liveness of pipelined Fast-HotStuff in a way similar to what we have proven for the basic Fast-HotStuff. For safety we want to prove that if a block is committed by a replica, then it will never be revoked. Moreover, if a replica commits a block then all other replica will eventually commit the same block at the same block sequence/height.

Lemma 5. *If B and B' are two conflicting blocks, then only one of them will be committed by a replica.*

Proof. From lemma 1, we know that each correct replica votes only once for each view and therefore view number for conflicting blocks $B.curView < B'.curView$ and we assume that block B is already committed. Now, a replica r receives a PREPARE message in the form of block B'. Since B' is a conflicting block to B therefore, the *highQC* in the block must point to an ancestor of block B. In this case we consider B^* as the parent of B ($B.parent = B^*$) and $B'.QC.block = B^*$. Since B' extends from B^* which is parent of B, therefore first condition in *PipelinedSafeBlock* predicate fails because *highQC* in B' ($B'.QC$) extends the parent of B (B^*) but not B. Similarly, second condition in *PipelinedSafeBlock* predicate also fails because $B'.viewNumber = B'.QC.viewNumber + k$, where $k > 1$. □

Algorithm 3: pipelined Fast-HotStuff for block i

```
1  foreachin curView ← 1,2,3,...
2      if i is primary then
3          if $n-f$ $\eta$ msgs are received then
4              $aggQC \leftarrow CreateAggQC(\eta_{Set})$
5              $B \leftarrow$ CreatePrepareMsg(Prepare, aggQC, $\perp$, client's command)
6          end
7          if $n-f$ $v$ msgs are received then
8              $qc \leftarrow GenerateQC(V)$
               CreatePrepareMsg(Prepare, $\perp$, qc, client's command)
9          end
10         broadcast $B$
11     end
12     if i is normal replica then
13         wait for $B$ from primary(curView)
14         if PipelinedSafeBlock($B, B.qc, \perp$) $\vee$ PipelinedSafeBlock($B, \perp, B.aggQC$) then
15             Send vote $v$ for prepare message to primary(curView+1)
16         end
17         // start commit phase on $B^*$'s grandparent if direct chain exists among $B^*$, its parent and $B^*$'s grandparent
18         if $(B^*.parent = B'' \wedge B*.view = B''.view+1) \wedge (B''.parent = B' \wedge B''.view = B'.view+1)$ then
19             execute new commands through $B'$
20             respond to clients
21         end
22     end
23     if i is next primary then
24         wait until $(n-f)$ $v/\eta$ for current view are received: $\eta_{Set} \leftarrow \eta_{Set} \cup \eta \vee V \leftarrow V \cup v$
25     end
26     ▷ Finally
27     check always for nextView interrupt then
28         goto this line if nextView(curView) is called during "wait for" in any phase
29         Send $\eta$ to primary($curView+1$)
30     end
```

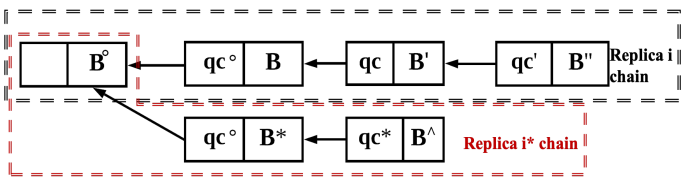


Fig. 5. **Safety violation scenario**.

Lemma 6. *If a block B is committed by a replica i at the height h, then no other replica i^* in the network will commit another block B^* at the height h.*

Proof. For the sake of contradiction let's assume that it is possible that if block B at the height h is committed only by a single replica i then at least one another replica i^* can commit another block B^* the height h.The chain of each replica i and i^* is given in the Figure 5. Now we analyze different cases arise from our assumption.

Case 1: When qc*.view $<$ qc.view. In this case, since $qc^*.view < qc.view$ therefore, $qc^*.view < highQC.view$. As a result, qc^* will not be selected as $highQC$ by any honest replica (based on Lemma 2).

Case 2: When qc*.view $==$ qc.view. Two QCs pointing to two different blocks cannot be same.

Case 3: When qc'.view $>$ qc*.view $>$ qc.view In the third case, there is a possibility that a number of replicas hold a QC qc^* such that $qc^*.view > qc.view$. qc^* has not been propagated to the majority of honest nodes due to the network partition. The $qc^*.block$ is conflicting block to block B. After the view change the new primary p (that is aware of qc^*) includes qc^* in the $AggQC$. If qc^* is accepted then it is possible that other replicas may commit a block B^* which is conflicting to block B. But this is not possible due to the two-direct chain requirements to commit a block ($qc.view+1 = qc'.view$) in Algorithm 3 lines 18-20. Therefore, there can be no such qc^*, with a view $qc'.view > qc^*.view > qc.view$ when replica i has committed the block B.

Case 4: When qc*.view $==$ qc'.view Two QCs pointing to two different blocks cannot be the same.

Case 5: When qc*.view $>$ qc'.view We know that qc' has been built from $n-f$ replica votes that have seen the qc. Therefore, the qc is the $highqc$ for $n-f$ replicas of which at least $f+1$ are honest. Hence, when the primary collect $n-f$ η messages (NEWVIEW) during view change at least one of η will contain qc or a QC that extends the qc (based on Lemma 2). Therefore qc^* that does not extend qc will not be formed. Hence, our assumption is proved to be false. Therefore, if a block is committed by a replica i at height h then no other block can be committed at the height h.

□

This lemma confirms that if a single replica commits a block B, then no conflicting block to B will be committed by another replica. This lemma is developed to address a subtle safety violation scenario due to the network partition that was reported in Gemini [21] (that simulates Byzantine scenarios at scale) by Facebook's Diem group (previously known as Novi). We also successfully verified the safety of revised Fast-HotStuff using the Twins simulator for Fast-HotStuff developed by Diem team[6].

E. Performance Penalty of direct chain Condition

Direct chain has a high cost in HotStuff protocol due to the forking attack. Each time when a Byzantine primary overrides blocks for honest primaries it not only reduces the throughput but also prevents direct chain formation. This results in increased latency. On the other hand, since a forking attack is not possible in HotStuff, a direct parent cannot be broken through forking. But it is possible that a direct chain is broken when a primary fails to propose a block and replicas timeout. Since replicas wait for the maximum timeout period Δ, which is very large than the actual wire speed δ ($\delta \ll \Delta$). Therefore,

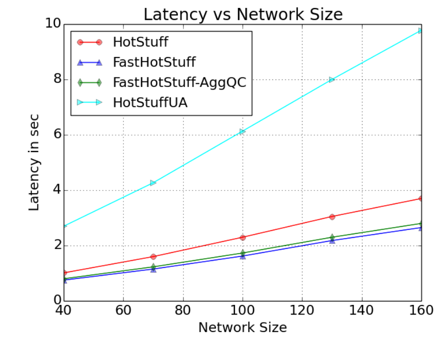
[6]https://github.com/asonnino/twins-simulator

incurring latency in the order of δ (due to lack of direct chain) is negligible when the overall delay is in the order of Δ.

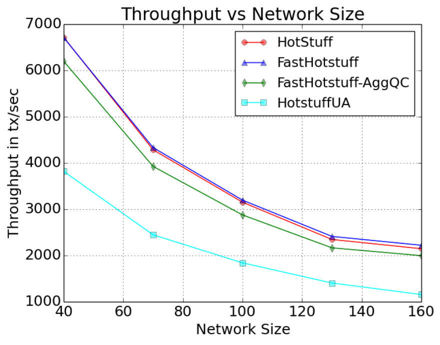
The liveness proof for pipelined Fast-HotStuff is similar to the proof for Basic Fast-HotStuff and is omitted here for brevity.

F. Communication complexity

We separate our analysis on communication complexity of Fast-HotStuff in happy path and unhappy path. In happy path, the communication complexity of pipelined Fast-HotStuff is the same with HotStuff, and both are $O(nl+nk)$ where l is the block payload size and k is the size of a vote or the NEWVIEW message. While in unhappy path (primary failure), the communication complexity of the pipelined Fast-HotStuff is $O(nl+n^2k)$. As in applications such as blockchain, the block size l is usually large (in megabytes), therefore, nl is the dominating component in communication complexity and n^2k has negligible effect on performance. That is why even in unhappy path, Fast-HotStuff still achieve a similar performances with HotStuff.



(a) Fast-HotStuff Latency

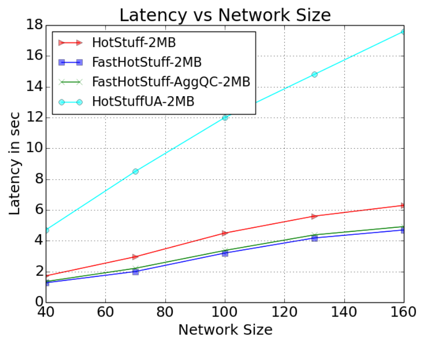


(b) Fast-HotStuff Throughput

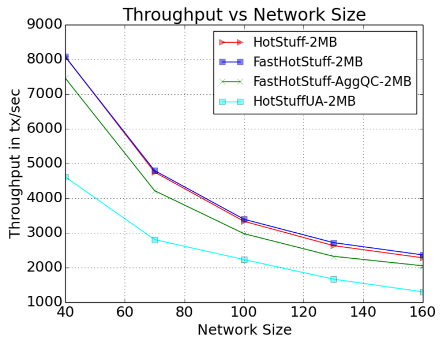
Fig. 6. Performance tests with 1MB block size

V. Evaluation

As pipelined HotStuff is widely adopted due to its higher throughput, we implemented prototypes of pipelined Fast-HotStuff and pipelined HotStuff using Go programming language. For cryptographic operations we used dedis advanced crypto library in Go [22] . Moreover, for hashing values, we



(a) Fast-HotStuff Latency



(b) Fast-HotStuff Throughput

Fig. 7. Performance tests with 2MB block size

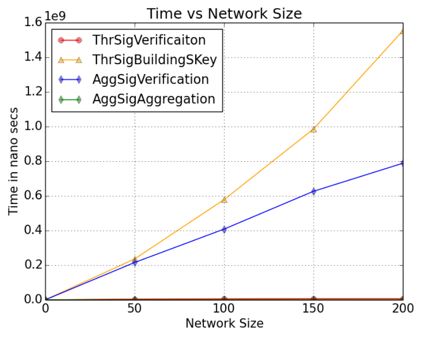


Fig. 8. **Comparing Signature Schemes**.

used SHA256 hashing in the Go crypto library [7]. We tested Fast-HotStuff's performance on Amazon cloud (AWS) with different network sizes $40, 70, 100, 130, 160$ (to compare performance and scalability) and different block sizes ($1MB$ and $2MB$). We used $t2.micro$ replicas in AWS, where each replica has a single vCPU (virtual CPU) comparable to a single core with $1GB$ memory. The bandwidth was set to 500 Mb/sec (62.5 MB/sec) and the latency between two end points was set to 50 msec. We also performed forking attack on each network of different sizes, $k = 1000$ times and took mean of the average throughput of the network and the latency incurred by blocks affected by this attack.

We compared pipelined Fast-HotStuff's performance (throughput and latency) against pipelined HotStuff in three

[7]https://golang.org/pkg/crypto/sha256/

scenarios: 1) during happy path (when no failure occurs), therefore, the primary only includes the *QC* in the block (red vs blue curves), 2) when the previous primary fails and the next primary in the Fast-HotStuff has to include the aggregated *QC* ($(n-f)$ *QC*s) in the block (red vs green curves), and 3) when HotStuff and Fast-HotStuff come under forking attack. Results achieved from each case are discussed below:

1) As the results shown in Figure 6a and 7a, during happy path when no failure occurs, Fast-HotStuff outperforms HotStuff in terms of latency. HotStuff's throughput (in red) slightly decreases against Fast-HotStuffs throughput (in blue) due to $O(n^2)$ time complexity for interpolation calculation at the primary when n increases (Figure 6b and 7b).
2) We did not consider the timeout period taken by replicas to recover, as our main objective is to show that even the inclusion of aggregated *QC* (small overhead) after a primary failure has a negligible effect on pipelined Fast-HotStuff's performance. Therefore, just after GST (after primary failure), when an honest primary is selected, HotStuff does not incur additional overhead. However, HotStuff needs an additional round of consensus during happy (normal) as well as unhappy (failure) cases in comparison to Fast-HotStuff. As a result, the throughput and latency of HotStuff during the happy path or unhappy path just after primary failure is the same and is shown by the red curve (by ignoring the timeout period). Consequently, HotStuff's throughput (in red) is slightly better than Fast-HotStuff's throughput (in green). See in Figure 6b and Figure 7b. Despite the small overhead, Fast-HotStuff's latency (in green) is lower than the HotStuff's latency (red) as shown in Figure 6a and Figure 7a.
3) Forking attack effects in Fast-HotStuff and HotStuff can be observed in Figure 6a and b and Figure 7a and b by comparing blue vs. cyan curves. Since forking attack has no effect on Fast-Hotstuf (as explained in subsection IV-F), there is no difference between Fast-HotStuff's normal throughput and latency and its throughput and latency under attack. As such, we did not use additional curves representing the performance of Fast-HotStuff under attack in Figure 6a and b and Figure 7a and b (blue). By contrast, under attack, HotStuff's latency increases and its throughput decreases significantly (shown with cyan curve) .

HotStuff uses *t-out-of-n* threshold signatures [16] and the Fast-HotStuff uses aggregated signatures as stated previously. The crypto library we used (Dedis advanced crypto library) supports both aggregated and threshold signatures. In order to better understand the effects of signature schemes on each protocol's performance we also compared the latency caused by each signature type during consensus. Higher latency cause by cryptographic operations during consensus will result in higher latency and lower throughput. In both aggregated signature as well as threshold signature there are three main steps for signature verification in HotStuff and Fast-HotStuff. First, each replica signs a message (vote, *NEWVIEW*) and sends it to the primary. Second, the primary builds an aggregated signature or threshold signature from $n-f$ messages received from distinct replicas and sends it back to each replica. Third, each replica verifies the aggregated or threshold signature associated with the messages received from the primary. The cost for the first step is small and constant (for constant size message) for both threshold signature as well as aggregated signature. Therefore, we ignore this cost. The computation cost of aggregating signatures into a single constant size aggregated signature by the primary replica is small, resulting in a negligible delay. It is shown by the green curve in the Figure 8. However, the cost of building threshold signature from $n-f$ partial signatures received by the primary replica from $n-f$ replicas (shown by the yellow curve) in the threshold signatures is quadratic as it uses $O(t^2)$ $(t = n-f)$ time polynomial interpolation. Conversely, the computation cost of verification in threshold signature is small and results in a constant delay per replica shown by the red curve. However, the cost of verifying aggregated signature in the aggregated signature scheme is linear, resulting in a linear delay shown by the blue curve. In summary, we can see that building threshold signatures in the primary from partial signatures is expensive in threshold signatures. Although threshold signatures have constant verification costs, the latency induced in the primary prevents threshold signatures from having any performance gain in comparison to the aggregated signature scheme.

VI. Related Work

There have been multiple works on improving BFT protocols performance and scalability [10], [13], [23]. But these protocols suffer from expensive view change where either the message complexity or a number of signatures/authenticators to be verified grows quadraticly. Moreover, these protocols do not employ a primary rotation mechanism.

Other protocols that offer a simple mechanism of leader/primary replacement include Casper [3] and Tendermint [1]. But both of these protocols have synchronous cores where replicas in the network have to wait for the maximum network latency before moving to the next round. In other words, these protocols lack responsiveness which will result in high-performance degradation.

PBFT [5] has quadratic message complexity during happy path or normal operation. During view change each replica has to process at least $O(n^2)$ signatures. Moreover, PBFT does not use rotating primary. On the otherhand, Fast-HotStuff uses rotating primary, pipelined Fast-HotStuff has linear view change during normal primary rotation (view change). In case of failure, primary rotation in Fast-HotStuff in each replica the processing cost of signatures is either linear or constant depending on signature scheme used.

SBFT [11] has linear communication complexity during normal operation. SBFT does not employ rotating primary mechanism. When Fast-HotStuff and SBFT use the same signature scheme, replicas in Fast-HotStuff will have to verify $O(n)$ signatures less than the SBFT.

HotStuff [4] is designed to not only keep a simple leader change process but also maintain responsiveness. These features along with the pipelining optimization have provided an opportunity for wide adoption of the HotStuff [7] protocol. HotStuff has linear view change but we show that in practice during primary failure (unhappy path) Fast-Hotstuff's view change performance is comparable to HotStuff. LibraBFT [7] is a variant of HotStuff. Unlike the HotStuff that uses threshold signatures, Libra BFT uses aggregated signatures. LibraBFT uses broadcasting during primary failure. LibraBFT also has three-chain structure and is susceptible to forking attacks.

Pala BFT [2] is another variant of HotStuff (though the paper has not been peer reviewed) that introduces a strong notion of synchrony. Every replica has to multicast any message it has seen, making the protocol highly expensive. Pala requires that the clocks in replicas are synchronous and with negligible bounded skew. Pala is also susceptible to forking attacks. On the other hand, Fast-HotStuff does not require any notion of synchronous clock and is highly event-driven once a block is proposed by the primary. This makes Fast-HotStuff a highly reliable and robust protocol that can safely be used over the Internet where message latency may not be always uniform. Fast-HotStuff is also robust against forking attack as mentioned previously.

VII. Conclusion

In this paper, we presented Fast-Hotstuff which is a two-chain consensus protocol with efficient and simplified view change. It achieves consensus in two rounds of communication. Moreover, Fast-Hotstuff is robust against forking attacks. Whereas HotStuff lacks resilience against forking attacks. Fast-Hotstuff achieves these unique advantages by adding a small amount of overhead in the block. This overhead is only required in rare situations when a primary fails. Our experimental results show that whether the overhead is included in the proposed block or not, Fast-Hotstuff outperforms Hotstuff in terms of latency. Fast-HotStuff outperforms HotStuff in terms of latency and throughput under forking attack.

Acknowledgement

The authors gratefully acknowledge valuable discussions with Dr. Dahlia Malkhi and her team which helped to improve this work. Specifically by finding a subtle safety violation through their automated testing tool for BFT consensus Gemini [21].

References

[1] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Jun 2016.

[2] T.-H. H. Chan, R. Pass, and E. Shi, "Pala: A simple partially synchronous blockchain," Cryptology ePrint Archive, Report 2018/981, 2018, https://eprint.iacr.org/2018/981.

[3] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[4] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM PODC*, ser. PODC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–356.

[5] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186.

[6] M. M. Jalalzai, C. Feng, C. Busch, G. G. R. I. au2, and J. Niu, "The hermes bft for blockchains," 2020.

[7] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino, "State machine replication in the libra blockchain," 2019.

[8] A. Hentschel, Y. Hassanzadeh-Nazarabadi, R. M. Seraj, D. Shirley, and L. Lafrance, "Flow: Separating consensus and compute - block formation and execution," *ArXiv*, vol. abs/2002.07403, 2020.

[9] Y. Guo, Q. Yang, H. Zhou, W. Lu, and S. Zeng, "Syetem and methods for selection and utilizing a committee of validator nodes in a distributed system," Cypherium Blockchain, Feb 2020, patent.

[10] M. M. Jalalzai and C. Busch, "Window based BFT blockchain consensus," in *iThings, IEEE GreenCom, IEEE (CPSCom) and IEEE SSmartData 2018*, July 2018, pp. 971–979.

[11] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu, "SBFT: a scalable decentralized trust infrastructure for blockchains," *CoRR*, vol. abs/1804.01626, 2018.

[12] R. Kotla, A. Clement, E. Wong, L. Alvisi, and M. Dahlin, "Zyzzyva: Speculative Byzantine fault tolerance," *Commun. ACM*, vol. 51, no. 11, pp. 86–95, Nov. 2008.

[13] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 355–362.

[14] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, vol. 00, June 2008, pp. 197–206. [Online]. Available: doi.ieeecomputersociety.org/10.1109/DSN.2008.4630088

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

[16] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ser. ASIACRYPT '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 514–532.

[17] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *Proceedings of the 22nd International Conference on Theory and Applications of Cryptographic Techniques*. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 416–432.

[18] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *Advances in Cryptology – ASIACRYPT 2018*, T. Peyrin and S. Galbraith, Eds. Cham: Springer International Publishing, 2018, pp. 435–464.

[19] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[20] A. , "Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme," in *Proceedings of the 6th International Workshop on Theory and Practice in Public Key Cryptography: Public Key Cryptography*, ser. PKC '03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 31–46.

[21] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Gemini: Bft systems made robust."

[22] P. Jovanovic, J. R. Allen, T. Bowers, and G. Bosson, "dedis kyber," 2020.

[23] M. M. Jalalzai, C. Busch, and G. G. Richard, "Proteus: A scalable bft consensus protocol for blockchains," in *2019 IEEE International Conference on Blockchain (Blockchain)*, 2019, pp. 308–313.