Abstract—As the first Byzantine fault-tolerant (BFT) protocol with linear communication complexity, HotStuff (PODC 2019) has received significant attention. HotStuff has three round-trips for both normal case operations and view change protocols. Follow-up studies attempt to reduce the number of phases for HotStuff. These protocols, however, all give up on one thing in return for another.

This paper presents Marlin, a BFT protocol with linearity, having two phases for normal case operations and two or three phases for view changes. Marlin uses the same cryptographic tools as in HotStuff and introduces no additional assumptions. We implement a new and efficient Golang library for Marlin and HotStuff, showing Marlin outperforms HotStuff for both the common case and the view change.

I. INTRODUCTION

Byzantine fault-tolerant state machine replication (BFT) is a fundamental tool in fault-tolerant distributed computing [16, 20, 21, 23, 25, 45]. BFT has nowadays gained growing attention, as it is the de facto model for permissioned blockchains [8, 11, 18, 43, 46, 51].

Being the first BFT protocol with linear communication complexity, HotStuff [52] has been used by the Diem blockchain platform. The technique underlying HotStuff has also proven significant, yielding novel protocols such as [3]–[5]–[8, 48].

Strikingly, HotStuff is best known as a BFT with linear authenticator complexity if instantiated using threshold signatures [12, 44], but its most efficient implementation is to instantiate threshold signature using a group (linear number) of standard signatures: multiple platforms and systems [2, 31, 40] have reported HotStuff with conventional signatures outperforms HotStuff with the most efficient threshold signature (or multi-signature), unless one tests a scenario that 1) has a significant network latency, where the cryptographic overhead is less visible, and 2) has a low network bandwidth and a large n (the number of replicas), where n signatures are no longer bandwidth negligible compared to operations [40]. The fact should not be surprising, as the most efficient dedicated threshold signatures use expensive cryptographic pairings. Computing pairings is at least an order or several orders of magnitude slower than signatures [10].

HotStuff commits operations in three round-trips (phases), but the optimal latency (for HotStuff-style protocols) is two phases. Naturally, there has been a line of works aiming at reducing the number of rounds for HotStuff, such as Fast-HotStuff [34], AAR [6], Jolteon/Ditto [30], and Wendy [31].

Brief review of two-phase BFT protocols. Fast-HotStuff [34] and Jolteon [30] have a two-phase normal case operation but have a quadratic communication overhead in the view change protocol. Diem Team is currently integrating Jolteon into the next release of DiemBFT [30] for its performance in normal case operations, which further motivates the study of two-phase BFT protocols with linearity.

Abspeol, Attema, and Rambaud (AAR) [6] propose a theoretical work that reduces the number of phases of HotStuff to two. AAR has a quasilinear communication cost (O(n log n)). The protocol, however, cannot be efficiently implemented, as it uses prohibitively expensive zero-knowledge proof systems.

A concurrent and beautiful system, Wendy [31], uses a novel aggregate signature scheme to build a new HotStuff-style protocol that has a two-phase normal case protocol and has at most three phases in the view change. The technique used is very interesting in the sense it leverages pairing-based cryptography to prove an operation did not commit. Wendy, however, has the following features: 1) Wendy introduces an additional assumption: the view number difference c between any replica and the leader is bounded and must be fixed in the system setup phase. The difference c (bounded by the view number bound u) is proportional to the size of the public keys and the cryptographic overhead. To be safe, c should be reasonably large. 2) Strictly speaking, Wendy does not achieve linear communication or linear authenticator complexity. During view change, the communication complexity can be O(n^2 log u + nλ), where u is the view number bound and λ is a security parameter, while the authenticator complexity is O(n^2) (see definitions in Section III for complexity measures of aggregate signatures and multi-signatures and see Section IV for an illustration). Note that the number of pairings needed is O(n) in a view change, but other public-key cryptographic operations (group multiplications) remains O(n^2 log c). 3) As reported by Wendy [31, Section VII.D], due to the usage of pairings, Wendy may have lower performance than HotStuff in view change. Wendy, however, has a very nice feature that when there are no attacks, the unhappy cases in view changes may be really rare.

Hence, all these HotStuff descendants make trade-offs: a more expensive or sometimes more expensive view change for a two-phase normal case. It remains an open question whether we can design a better HotStuff-style BFT protocol without making a trade-off. (In fact, it is also an open problem whether one could design BFT with a two-phase commit in normal cases and a linear communication in view changes.)
Our contributions. We make the following contributions:

- We provide a new HotStuff-style BFT protocol—Marlin. Marlin achieves strictly linear communication complexity and authenticator complexity, having two phases in normal-case operations and two or three phases in view changes. In the "happy" path, Marlin has two phases for a view change. Besides, Marlin is well compatible with the chaining (pipelining) technique. In contrast, all other HotStuff variants all give up one thing for another and, strictly speaking, none of them have linear communication or linear authenticator.
- We introduce the notion of view change snapshot to explain and analyze HotStuff and its variants. The notion unifies the theory of existing approaches and facilitates designing new protocols.
- We develop new techniques to build HotStuff-style BFT, including a new way of unlocking a locked block, introducing virtual blocks for early commits, and using shadow blocks to reduce bandwidth.
- We provide a new Golang library for Marlin and HotStuff. We have performed extensive evaluations using commodity servers. We show that, unlike all other HotStuff variants that are slower than HotStuff in many cases, Marlin outperforms HotStuff consistently.

II. RELATED WORK

Characterizing BFT protocols. BFT protocols can be roughly divided into two categories according to timing assumptions: asynchronous BFT and partially synchronous BFT. Asynchronous BFT protocols rely on no timing assumptions. Safety of partially synchronous BFT is always preserved, but liveness relies on the partial synchrony assumption [27]. A large number of partially synchronous BFT (e.g., [7, 16, 22, 24, 25, 32, 33, 45, 47, 49, 50]) and asynchronous BFT protocols (e.g., [9, 14, 15, 19, 26, 36, 38, 39]) have been proposed.

No one-size-fits-all BFT. While the paper (and other recent papers mentioned) advocate HotStuff-like BFT protocols, readers should be aware that there is no one-size-fits-all BFT protocol, even if we only consider the partial synchrony model. First, HotStuff has an end-to-end (client-to-client) latency of 9, while PBFT has a latency of 5. The two-phase variants of HotStuff, including Marlin, have a latency of 7. Second, as we have commented, there is a mismatch between authenticator complexity and practical implementations: the most efficient instantiation for HotStuff (for most cases) uses signatures and has $O(n^2)$ authenticator complexity. Third, it is unclear which of the two following is more robust: the linear communication that HotStuff uses, or the classic, broadcast-based communication that PBFT and other protocols adopt. For instance, some performance attacks seem to be HotStuff exclusive [29, 41].

The HotStuff techniques. Cachin, Kursawe, Petzold, and Shoup (CKPS) [14] uses threshold signature to build communication-efficient consistent broadcast, a primitive that is proposed by Reiter [42]. CKPS consistent broadcast includes 1) a dissemination phase that broadcasts some message and 2) an aggregation phase that collects proofs that the message has been received in the form of partial threshold signatures and then combines them to generate a threshold signature (a proof that is publicly verifiable). The HotStuff technique may be viewed as one using two or more CKPS consistent broadcast communication phases. The proof in the second phase can be used to prove succinctly a non-equivocating value has been accepted in the first phase. Further, the HotStuff technique may also be referred to as the "lock-commit-unlock" paradigm: replicas may become "locked" on a value when the value may have been committed by some other replica and can later unlock when the value did not commit. HotStuff techniques have been proven fruitful [3–5, 48].

Kauri. Kauri [40] is a new BFT communication abstraction that uses pipelining and tree-based dissemination and aggregation to achieve scalability. Kauri instantiates the framework using HotStuff. It makes sense to use two-phase BFT protocols in the Kauri framework for better performance.

Formal verification. Jehl [35] recently provides a formal verification for HotStuff using TLA [17] and Ivy [37].

III. SYSTEM MODEL

BFT. We consider a Byzantine fault-tolerant state machine replication (BFT) system consisting of $n$ replicas, where $f$ of them may fail arbitrarily (Byzantine failures). We require $n \geq 3f + 1$. In BFT, a replica delivers or commits client operations submitted by clients. A replica then sends a reply to the corresponding client. The client computes a final response based on the reply messages. We consider the partially synchronous model [28], where there exists an unknown global stabilization time (GST) such that after GST, messages sent between two correct replicas arrive within a fixed delay.

Cryptographic primitives. We use the definition of [12, 44] for a $(t, n)$ threshold signature scheme consisting of the following algorithms (tgen, tsign, tcombine, tverify). tgen outputs a system public key known to anyone and a vector of $n$ private keys. A partial signature signing algorithm tsign takes as input a message $m$ and a private key $sk_i$ and outputs a partial signature $\sigma_i$. A combining algorithm tcombine takes as input $pk$, a message $m$, and a set of $t$ valid partial signatures, and outputs a signature $\sigma$. A signature verification algorithm tverify takes as input $pk$, a message $m$, and a signature $\sigma$, and outputs a bit. We require the conventional robustness and unforgeability properties for threshold signatures. We may leave the verification of partial signatures and threshold signatures implicit when describing these algorithms. In this paper, we set $t$ to $n - f$.

Efficient instantiations for threshold signatures can be based on pairings [12, 13]. As we discussed in the introduction, one can use a group of $n$ signatures to build a $(t, n)$ threshold signature for better efficiency for real case deployments. Following prior works, this paper assumes pairings for threshold signatures when considering complexity measures.

We use a collision-resistant hash function $h$ mapping a message of arbitrary length to a fixed-length output. We assume the length of all the above primitives (signatures and hashes) is $O(\lambda)$, where $\lambda$ is the security parameter.
Some BFT protocols mentioned in the paper (e.g., FastHotStuff, Wendy) use aggregate signature [12, 13] which allows anyone to aggregate signatures for different messages into a single aggregate signature. An aggregate signature for t messages and t public keys may be of the form \((m_1, m_2, \ldots, m_t, \sigma, pk_1, pk_2, \ldots, pk_t)\). Verifying an aggregate signature takes as input all t messages and t public keys.

**Complexity metrics.** This paper considers communication complexity, authenticator complexity, and cryptographic operations needed. Communication complexity means the total bits transmitted for all replicas. Authenticator complexity means the total number of authenticators received by all replicas. An authenticator in our constructions may be a signature, a partial signature, or a threshold signature.

Note for systems using aggregate signatures, one cannot claim that a single aggregate signature for t different messages of size L and t public keys is a single authenticator: the communication of transmitting such an aggregate signature is at least \(tL + \lambda + n\) (if using a n-size bit-vector to represent public keys), and the number of cryptographic operations needed is at least \(O(t)\) (for one has to take as input t public keys). Hence, we view an aggregate signature for t different messages as t authenticators. Note an aggregate signature for the same message, also called multi-signatures, may be characterized as a single authenticator.

Crucially, pairing operations are much more expensive than conventional public-key cryptographic operations (e.g., elliptic curve); thus, it is important to distinguish pairing operations from conventional (non-pairing) operations.

In general, communication complexity and the number of cryptographic operations are more precise measures than authenticator complexity alone. We consider all three measures.

### A. BFT consensus over graph of blocks

We extend the syntax of BFT replication over graphs modeled in HotStuff [52] for our purpose. The protocols considered in the paper are leader-based, proceeding in a succession of views numbered by monotonically increasing view numbers, and associating each view with a leader. The unique leader in each view \(v\) is denoted as \(L_v\). The most current view number maintained by a replica is denoted as \(cv\). In each view, replicas reach consensus on a sequence of blocks until a view change occurs. During the view change, a new leader is elected and meanwhile a new view starts.

Each replica stores a tree of blocks (nodes). A block \(b\) contains a parent link \(pl\), a batch of operations \(op\)’s, and their metadata. A parent link for \(b\) is a hash digest of its parent block. A branch led by a given block \(b\) is the path from \(b\) all the way to the root of the tree (called the genesis block). We define \(view\) for \(b\) as the view during which \(b\) is proposed. We define \(height\) for \(b\) as the number of blocks on the branch led by \(b\). The metadata for a block \(b\) include the \(view\) for \(b\), the \(view\) for its parent block, and the \(height\) for \(b\).

Note that one difference between our syntax and the syntax of HotStuff is that the block in our model includes the view number of its parent block. Also, in our syntax, multiple blocks with increasing heights, instead of a single block, can be delivered in the same view ("normal case operation").

For a BFT protocol, a monotonically growing branch becomes committed. Each time, a block extends the branch led by its parent block. We say a block \(b’\) is an extension of a block \(b\) if \(b\) is on the branch led by \(b’\). We say two branches are conflicting, if neither one is an extension of the other. We say two blocks are conflicting, if the branches led by them are conflicting.

We use Figure 1 to illustrate our notation. \(b_1\) is committed in view 1, while \(b_2\) and \(b_3\) are committed in view 2. A branch led by \(b_2\) is the path from \(b_2\) to \(b_0\). \(b_3\), for instance, is an extension of \(b_2\) and also an extension of \(b_1\). The height of \(b_3\) is 4, equal to the depth of the tree. The parent link for a block \(b_2\) is a hash of its parent block \(b_1\). \(b_3\) and \(d_3\) are conflicting, as the branches lead by them are conflicting.

With the syntax of BFT over graphs, we can recast the safety definition of BFT as follows: no two correct replicas commit two conflicting blocks. The liveness definition requires that after GST, any operation proposed by a client will be eventually executed.

### IV. Review of Two-Phase BFT Protocols and Overview of Marlin

#### A. Review of HotStuff

HotStuff (in the rotating leader mode) delivers a proposal in each view (height) and adopts a three-phase commit rule: prepare phase, precommit phase, and commit phase. Each phase uses a threshold signature to achieve linear communication. In the prepare phase, the leader broadcasts a proposal (a block with some view) to all replicas and waits for signed responses (in the form of partial signatures) from a quorum of \(n - f\) replicas to form a threshold signature as the prepare quorum certificate (QC)—prepareQC. In the precommit phase, the leader broadcasts prepareQC and waits for responses to form precommitQC. Note the two phases can achieve safety but not liveness (when the leader is faulty). So a third phase is needed to broadcast precommitQC and form commitQC: after receiving precommitQC, replicas become locked on the QC and will not accept a conflicting block with the same view; replicas may later unlock only if they are shown with a QC of a higher view. After forming commitQC, the leader forwards it to all replicas that then safely deliver the proposal.

In view changes, each replica sends its latest prepareQC to the leader. After receiving a quorum of view change messages, the leader selects the QC with the largest height, extends the block for the QC with a new proposal.
B. Challenges of Building Two-phase HotStuff

To illustrate the challenges of reducing the phases of HotStuff, we begin with "two-phase HotStuff (insecure)" that has a two-phase commit rule. The first phase is the prepare phase with $prepareQC$, while the second phase is commit phase with $commitQC$. In particular, after receiving $prepareQC$, replicas become locked on the QC. The issue for the above approach is the QC the new leader obtains from a quorum of view change messages may not be the most recent QC generated before view change. In other words, replicas may be locked on a QC that the leader is not aware of. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.

We use the view change snapshots in Figure 2 to explain the liveness issues. These examples all consider a four-replica setting, where $p_2$ is the new leader, and $p_4$ is faulty and may choose to hide its latest QC. This scenario leads to liveness problems. To illustrate the problems, we define a notion of view change snapshot to denote a quorum of view change messages (containing QCs) a leader collects. Clearly, the leader may obtain different snapshots during view change. Our analysis should thus cover all possible scenarios for view change snapshots.

We say a view change snapshot is safe, if it includes the most recent $prepareQC$ before view change. Otherwise, we say a snapshot is unsafe.
Case 1: The leader receives (non-pairing operations): leader; case comparison of HotStuff and two-phase variants. Here, vc stands for view change. The value broadcasts a pre-prepare phase. Wendy may be slower than HotStuff in view change.

Depending on the QC it is locked on. Interestingly (and not (Case 2). Each replica can either vote for one or two blocks (that may or may not exist) from a "virtual," safe snapshot that occurs, a pre-prepare phase, does not know which is not "wasted." The apparent obstacle is that the leader, at after the pre-prepare phase to preserve liveness for the case anything better, as one has to commit a block in three phases however, doing so will lead to a linear view change protocol prepareQC is formed; Case 2): a higher prepareQC exists and has been voted by a quorum of replicas to form qc. In other words, the virtual block now has a "real" parent block. The leader uses the prepareQC and the higher prepareQC to validate the virtual block.

Figure 2c presents an example, where the leader p1 receives QC for block b1. As p2 is unsure if there exists a higher QC, it can simply propose b2 that extends b1 and b2′ that extends a nil block. Here, p1 only votes for b2′, while p2 and p3 vote for both b2 and b2′. After receiving the votes from p1 and p3 and a prepareQC from p1, p2 can form a valid prepareQC for b2′. But a prepareQC for b2 cannot be formed due to the lack of the vote from p1. Even if a faulty leader may, in extreme cases, collect two prepareQCs, the scenario can be handled by our protocol (in the following prepare phase). Shadow blocks. One drawback for the above approach is that the prepare phase proposes two blocks but only decides one block eventually. In Marlin, when the leader proposes two blocks in this phase, the leader chooses the blocks that have the same operations but different associated metadata, thereby saving bandwidth.

V. THE MARLIN PROTOCOL

Marlin has two phases for normal operations (prepare phase and commit phase) and three phases for view changes (prepare phase, prepare phase, and commit phase).

A. Marlin-Specific Data Structures

Normal blocks and virtual blocks. A block is represented in the form b = [pl, pview, view, height, op, justify], where pl is the hash of the parent block of b, pview is the view number of the parent block of b, view is the view number of b, height is the height of b, op is a batch of client operations, and justify contains a quorum certificate (QC) for the parent block of b. We will use b.x to denote the element x of b. In a normal block, all fields should be specified. We also define a virtual block, a special block used in view change. It differs from a normal block in that its pl field is set to ∃.

Message format. A message m contains several fields: m.view, m.type, m.block, m.justify, and m.parsig.

<table>
<thead>
<tr>
<th>protocol</th>
<th>vc communication</th>
<th>vc cryptographic operations</th>
<th>vc authenticator</th>
<th>vc # phases</th>
</tr>
</thead>
<tbody>
<tr>
<td>HotStuff [52]</td>
<td>O(nλ + n log u)</td>
<td>O(n²) non-pairings or O(n) pairings</td>
<td>O(n)</td>
<td>3</td>
</tr>
<tr>
<td>Fast-HotStuff [34]</td>
<td>O(n²λ + n² log u)</td>
<td>O(n) non-pairings or O(n²) pairings</td>
<td>O(n²)</td>
<td>2</td>
</tr>
<tr>
<td>Jolteon [30]</td>
<td>O(nλ + n² log u)</td>
<td>O(n²) non-pairings or O(n²) pairings</td>
<td>O(n²)</td>
<td>2</td>
</tr>
<tr>
<td>Wendy [31]</td>
<td>O(nλ + n log u)</td>
<td>O(n²λ) non-pairings and O(n) pairings</td>
<td>O(n)</td>
<td>2 or 3</td>
</tr>
<tr>
<td>Marlin (this work)</td>
<td>O(nλ + n² log u)</td>
<td>O(n²) non-pairings or O(n) pairings</td>
<td>O(n)</td>
<td>2 or 3</td>
</tr>
</tbody>
</table>

TABLE I: Comparison of HotStuff and two-phase variants. Here, vc stands for view change. The value u is the upper bound on the view number; log u is the size of the message space for u. c = O(u). The cryptographic operations distinguish between pairing operations and conventional public key cryptographic operations (non-pairing operations): or means operations instantiated using signatures only or threshold signatures only. Note Wendy relies on both ("and"). While Wendy in view change has O(n²) authenticators, it uses O(n) pairings. As reported by Wendy, due to the use of pairings, Wendy may be slower than HotStuff in view change.
m.view is the view in which m is sent. m.type ∈ \{new-view, pre-prepare, prepare, commit\}. m is called a message for block b, if m.block = b. m.parsig contains a partial signature and m.justify includes one or two QCs.

Quorum certificates. A quorum certificate (QC) is a threshold signature of a message m for a block b. Given a quorum certificate qc for m, type(qc) is m.type and block(qc) is m.block. We use qc.x to denote the element x of m.block.

Rank of QCs and blocks. We introduce a notion of rank (being inspired by but differing from \cite{30}) that can simplify our description. Intuitively, the rules of ranks help determine if a new proposal can be safely accepted. In the common case, ranks equal heights; more complex rank rules are used in the view change. Each QC qc has a rank, denoted as rank(qc). The rank(qc) does not implicitly return a value. Instead, we only care if the rank of a QC is higher than that of another one.

The rank takes as input qc.view, type(qc), and qc.height. The comparison rules are shown in Figure 4. If neither rank(qc1) > rank(qc2) nor rank(qc2) > rank(qc1), then rank(qc1) = rank(qc2).

Fig. 4: Rank comparison rules.

Figure 5 presents an example of ranks. According to rule (a), rank(qc3) > rank(qc2). According to rule (b), rank(qc4) > rank(qc3) and rank(qc3) > rank(qc2). According to (c), rank(qc2) > rank(qc1). qc3 and qc2 have the same rank, although their heights are different.

We also define rank of blocks. For any two blocks b1 and b2, we say rank(b1) > rank(b2), if b1.view > b2.view or (b1.view = b2.view, b1.height > b2.height, and b1.justify is a prepareQC qc such that qc.view = b1.view).

B. Normal Case Protocol

Figure 6 and Figure 7 describe the pseudocode and the communication pattern, respectively. Each replica pi maintains four local variables: its current view cview, the last voted block lblock, lockedQC, and highQC. In particular, highQC stores QCs to be sent in VIEW-CHANGE messages.

Prepare phase. We distinguish two cases, where Case N1 corresponds to actions for successive normal case operations (with no view changes), and Case N2 corresponds to actions after the pre-prepare phase in view change. Right now, readers only need to understand Case N1 and should skip Case N2. Case N1 actions are similar to other two-phase HotStuff variants. Case N2 actions will become clear when we describe the view change protocol in Sec. V-C.

- Case N1: highQC is a prepareQC for a normal block b’. ▷ The leader lv proposes a new block b, where b.view = cview, b.pl is h(b’), b.height = h(b’).height + 1, op includes a batch of client operations, and b.justify is highQC.
- Case N2: highQC is a pre-prepareQC qc for a normal block or highQC is of the form (qc, vc).
  - Block b is set to block(qc).

Then lv broadcasts a prepare message m for b, where m.view and m.justify are set to cview and highQC.

After receiving a prepare message m from lv, such that m.block = b, replica pi verifies whether the message is well-formed, the proposal is created in the same view as cview, and b has a higher rank than its last voted block lb. Then pi verifies if m.justify is valid according to its local lockedQC by checking if one of the following holds:

- Case N1: m.justify is a prepareQC qc.
  ▷ pi checks whether b extends block(qc), vc.view = cview and rank(qc) ≥ rank(lockedQC).
- Case N2: m.justify is a pre-prepareQC qc for a normal block or m.justify is of the form (qc, vc).
  ▷ pi checks whether b is the same with block(qc), vc.view = cview and rank(qc) ≥ rank(lockedQC).
If m.justify is of the form (qc, vc), pi additionally verifies if qc is a pre-prepareQC for a virtual block and validate qc by verifying whether vc is a prepareQC, vc.view = qc.view, and vc.height = qc.height - 1.

Then pi sends lv a prepare message m’ for m.block together with a partial signature for m’. Meanwhile, pi updates its lb to m.block and sets highQC to m.justify. If m.justify is a prepareQC, pi sets lockedQC to m.justify.

Commit phase. Upon receiving n − f signed responses for the prepare message for b, the leader lv combines the partial signatures to form a prepareQC qc. Then lv broadcasts a commit message m for b, where m.justify = qc.

After receiving a valid COMMIT message m from lv, replica pi verifies whether the prepareQC included in the message is generated in current view. Then pi sends to lv a signed response for the COMMIT message for m.block. Replica pi also updates its highQC and lockedQC to m.justify.

Upon receiving n − f signed responses for the COMMIT message for b, lv forms a commitQC and forwards it to all replicas that then commit block b and its ancestors.

C. View Change Protocol

A view change is the mechanism through which a leader is replaced. A timeout is started when a replica enters a new view and a view change is triggered after the value expires. The pseudocode of the view change protocol is shown in Figure 9.

To start a new view v, each replica pi sets cview ← cview + 1 and sends a VIEW-CHANGE message m to current leader, where m.block is lb, m.justify is highQC, and m.parsig is a partial signature for lb.

Pre-prepare phase. The view change protocol begins with a pre-prepare phase. This phase is the most interesting and complex part of our protocol, and we describe it in full detail—distinguishing the code for the leader and replicas.
Let \( l_v \) be the leader of view \( v \). Each replica \( p_i \) keeps track of four variables: the current view \( cview \), the last voted block \( lb \), highQC, and lockedQC. Replicas initialize \( cview \leftarrow 1 \), \( lb \leftarrow \perp \), highQC \leftarrow \perp \), and lockedQC \leftarrow \perp .

**Normal case for replica \( p_i \):**
- **PREPARE.** (i) As a leader: broadcast a block \( b = [pl, pview, cview, height, ap, highQC] \) in a PREPARE message \( m \), where \( m.justify \) is \( highQC \), and \( b \) extends the block of \( highQC \) (Case N1) or \( b \) is the block of \( highQC \) (Case N2). (ii) As a replica: Upon receiving a valid PREPARE message from \( l_v \), if \( m.block \) has a higher rank than \( lb \) and \( m.justify \) is valid according to lockedQC, send \( l_v \) a signed response for the PREPARE message, and set \( lb \) to \( m.block \) and highQC to \( m.justify \). If \( m.justify \) is a prepareQC, set lockedQC to \( m.justify \).
- **COMMIT.** (i) As a leader: Upon receiving \( n - f \) signed responses for \( m \), form a prepareQC qc for \( b \) and broadcast a COMMIT message \( m \) for \( b \) where \( m.justify = qc \). Then wait for \( n - f \) signed responses to form a commitQC for \( b \) and forward it to all replicas. (ii) As a replica: Upon receiving a valid COMMIT message from \( l_v \), if \( m.justify.view = cview \), send \( l_v \) a signed response for the COMMIT message and set \( highQC \) and lockedQC to \( m.justify \). Then wait for a commitQC to commit \( b \) and its ancestors.

**Fig. 6:** Normal case operation for Marlin.

![Diagram](image.png)

**Fig. 7:** Normal case operation.

As a leader: Upon receiving a quorum of VIEW-CHANGE messages \( M_v \) for view \( v \), \( l_v \) begins the pre-prepare phase. In particular, \( l_v \) selects \( highQCV_v \)—valid QC(s) with the highest rank included in \( M_v \). In \( highQCV_v \), there may be one or two such QCs with the same (highest) rank. Also, let \( b_v \) be a block with the highest rank contained in the block field of \( M_v \). We distinguish three cases:

- **Case V1** (Figure 8a): \( highQCV_v \) is a prepareQC qc and at least one replica has voted for a block with a higher rank than qc (i.e., \( rank(b_v) > rank(block(qc)) \)).
  - \( l_v \) proposes two blocks: a normal block \( b_1 \) extending block(qc) and a virtual block \( b_2 \). height and \( b_2 \).pview are set to \( qc.height + 2 \) and \( qc.view, b_1 \).justify and \( b_2 \).justify are both set to qc. Then \( l_v \) broadcasts a PRE-PREPARE message with two proposals \( m_1 \) (for \( b_1 \)) and \( m_2 \) (for \( b_2 \)).
- **Case V2** (Figure 8b): \( highQCV_v \) is a prepareQC qc and \( rank(block(qc)) \) \( \geq \) \( rank(b_v) \), or \( highQCV_v \) contains only one valid prepareQC qc.
  - \( l_v \) proposes a normal block \( b \) which extends block(qc).
    - Then \( l_v \) broadcasts a PRE-PREPARE message \( m \) for \( b \).
- **Case V3** (Figure 8c): \( highQCV_v \) contains two valid prepareQCs qc1 and qc2.
  - \( l_v \) proposes two blocks: \( b_1 \) that extends block(qc1) and \( b_2 \) that extends block(qc2). Then \( l_v \) broadcasts a PRE-PREPARE message with two proposals \( m_1 \) (for \( b_1 \)) and \( m_2 \) (for \( b_2 \)).
  
  For each block \( b \), if a pre-prepareQC for a virtual block \( b' \) is included in \( b.justify \), the prepareQC for the parent block of \( b' \) should also be included in \( b.justify \) for verification. For each proposal \( m_i \), \( m_i,justify \) is set to \( m_i,block,justify \).

Then \( l_v \) waits for \( n - f \) signed responses for the PRE-PREPARE message to form a pre-prepareQC qc. If qc is a QC for a normal block, \( l_v \) sets highQC to qc; if qc is for a virtual block and meanwhile a prepareQC vc with a higher rank than highQC is received, \( l_v \) sets highQC to (qc, vc). Note that Case V1 and Case V3 propose two blocks. To reduce bandwidth, we ask them to be shadow blocks: they share the same bulk data (client operations) but differ in associated data only (e.g., height), so only one of them needs to carry operations.

As a replica: Each correct replica may vote for one or two proposals from \( l_v \). For each proposal \( m_i \), let \( q_c \) be the QC with the highest rank in \( m_i.justify \). Let \( b \) and \( b' \) denote \( m_i,block \) and block(qc), respectively. Replica \( p_i \) checks whether \( b \) extends block \( b' \) or \( b \) is a valid virtual block. If either case is satisfied, we distinguish three cases:

- **Case R1:** If \( q_c \) is a valid QC, \( rank(q_c) \geq rank(lockedQC) \), and \( q_c.view < cview \), then \( p_i \) sends \( l_v \) a PRE-PREPARE message \( m_i' \) together with a partial signature.
- **Case R2:** If \( m_i,justify = q_c, type = prepare, q_c.view < cview, q_c.view = lockedQC.view, q_c.height = lockedQC.height - 1 \), and \( b \) is a valid virtual block, then \( p_i \) sends \( l_v \) a PRE-PREPARE message \( m_i' \) together with a partial signature, where \( m_i',justify \) is set to lockedQC.
- **Case R3:** If \( q_c \) is a valid pre-prepareQC, \( q_c.view < cview \) and block(qc) = block(lockedQC), then \( p_i \) sends \( l_v \) a PRE-PREPARE message \( m_i' \) together with a partial signature.

**Prepare and commit phases.** After the pre-prepare phase, replicas switch to the prepare phase of the normal case and now it becomes clear that Case N2 in Sec. V-B for the normal case operation (code highlighted in gray) applies. In this case, the leader has just obtained a valid pre-prepareQC for a block and will send a PREPARE message for the block; replicas need to verify the pre-prepareQC included in the received PREPARE message. If the PREPARE message is formed for a virtual block \( b \), replicas have to additionally verify \( vc \) (a prepareQC) such that block(vc) is the parent block of \( b \). Furthermore, in Case N2, replicas do not become locked on a pre-prepareQC. (Otherwise, successive view changes may create liveness issues in a way much like the insecure two-phase HotStuff we analyzed.) In view change, replicas are locked on the prepareQC for block \( b \) in the commit phase and commit block \( b \) after receiving the corresponding commitQC.

**Ranks vs. heights.** Besides heights, we also define the rank (for both QCs and blocks) that is additionally related to the view number and the type of QCs. First, we define the rank for QCs to simplify the description of safety rules in view change. Second, we define the rank for blocks to enforce replicas to vote for only one block in the prepare and commit phases: note replicas can vote for two blocks during the view change, likely forming two pre-prepareQCs with the same rank; we
the pre-prepare phase can be skipped: if the new leader receives a prepareQC and is unsure whether it has a safe view change snapshot. This case occurs when the leader receives a prepareQC and is unsure whether it has a safe view change snapshot.

Fig. 8: Running examples for the pre-prepare phase in the view change.

![Diagram](image)

Replica $p_l$ switches to the view change protocol if timeout occurs in any phases. Replica $p_l$ sets $cview ← cview + 1$ and sends its VIEW-CHANGE $m$ for view $cview$ to the current leader, where $m$.block ← $lb$ and $m$.justify ← highQC.

**View change for replica $p_l$ in view $v$:**

- **PRE-PREPARE.** (i) As a leader: Upon receiving $M_e$ (a set of $n − f$ VIEW-CHANGE messages for view $v$), let $highQC_v$ be the valid QC(s) with the highest rank contained in the justify field of $M_e$ and $b_v$ be (any) one block with the highest rank in the block field. We distinguish three cases:
  - **Case V1:** $highQC_v$ is a prepareQC $qc$ and $rank(block(qc)) < rank(b_v)$. Broadcast a PRE-PREPARE message $m = m_1 || m_2$, where $m_1$ is a proposal for a normal block $b_1 = [hash(b_1'), b_1.view, cview, b_1.height + 1, op, highQC_v]$ and $m_2$ is a proposal for a virtual block $b_2 = [v, b_2.view, cview, b_2'.height + 2, op, highQC_v]$ such that both $m_1$.justify and $m_2$.justify are set to highQC_v.
  - **Case V2:** 1) $highQC_v$ is a prepareQC $qc$ and $rank(block(qc)) ≥ rank(b_v)$ or 2) $highQC_v$ contains one valid prepareQC $qc$. Let $b'$ denote $block(highQC_v)$. Broadcast a PRE-PREPARE message $m$ for block $b = [hash(b'), b'.view, cview, b'.height + 1, op, highQC_v]$, where $m$.justify is highQC_v.
  - **Case V3:** $highQC_v$ contains two valid prepareQCs $qc_1$ (for a normal block) and $qc_2$ (for a virtual block): Let $b_1'$ and $b_2'$ denote $block(qc_1)$ and $block(qc_2)$, respectively. Broadcast a PRE-PREPARE message $m = m_1 || m_2$, where $m_1$ is a proposal for block $b_1 = [hash(b_1'), b_1'.view, cview, b_1'.height + 1, op, qc_1]$ and $m_2$ is a proposal for block $b_2 = [hash(b_2'), b_2'.view, cview, b_2'.height + 1, op, qc_2]$. In $m_2$, $vc$ is the prepareQC for the parent block of $b_2$.

   □ Wait for $n − f$ signed responses for a PRE-PREPARE message, form a prepareQC, update highQC and switch to prepare phase.

(ii) As a replica: Upon receiving from $l_v$ a valid PRE-PREPARE message $m$ that may include one or two proposals ($m_1$ and $m_2$), if for each such proposal $m_i$, $m_i$.justify is formed before view $v$, then do the following:
  - **Case R1:** If $m_i$.justify includes a valid QC $qc$ and $rank(qc) ≥ rank(lockQC)$, send $l_v$ a signed response for $m_i$.
  - **Case R2:** If $m_i$.justify is a valid prepareQC $qc$, view = lockQC.view and $m_i$.block is a virtual block with height lockedQC.height + 1, send $l_v$ its lockQC and a signed response for $m_i$.
  - **Case R3:** If $m_i$.justify includes a valid prepareQC $qc$ and block($qc$) = block(lockQC), send $l_v$ a signed response for $m_i$.

**Fig. 9:** View change for Marlin.

thus additionally track ranks of these blocks to trivially handle the "forking" issue in the following prepare phase.

**Happy path in view change.** So far we have described the protocol where the pre-prepare phase is needed for a three-phase view change. There is, however, a happy path such that the pre-prepare phase can be skipped: if the new leader receives $n − f$ VIEW-CHANGE messages with the same $lb$, $l_v$ can combine the partial signatures into a prepareQC and directly switch to the prepare phase. Hence, the view change in Marlin may have two or three phases.

**Chained Marlin.** As in HotStuff and all its descendants, Marlin fully supports the chaining (pipelining) mode. Note that no new block is proposed in the prepare phase immediately after the pre-prepare phase in an unhappy view change. The feature happens to be similar to Wendy.

**D. Proof of Correctness**

We provide a proof of correctness for Marlin assuming the optimal resilience of $n = 3f + 1$. For safety, we first prove that Marlin is safe within a view and across views. For liveness, we prove that Marlin achieves liveness after GST.

**Lemma 1.** Let $b_1$ and $b_2$ be two blocks proposed in view $v$ such that the view of the parent block of $b_1$ (denoted $b_1'$) and the view of the parent block of $b_2$ (denoted $b_2'$) are lower than $v$. If the prepareQCs for $b_1$ and $b_2$ are both formed in view $v$, then $b_1 = b_2$ and prepareQC for $b_1$ is the prepareQC with lowest height formed in view $v$.

Proof. Let $b''$ be the block with the lowest height for which a prepareQC was formed in view $v$. If the parent block of $b''$ is proposed in view $v$, $b''$.justify should be a prepareQC for its parent block, contradicting the definition of $b''$. Thus, the view of the parent block of $b''$ is lower than $v$ and $b''$.justify is formed in a view lower than $v$. Hence, $rank(b_1) = rank(b_2) = rank(b'')$. As the prepareQCs for $b_1$, $b_2$, and $b''$ are all formed in view $v$, at least a correct replica has voted for $b_1$, $b_2$, and $b''$ in the prepare phase. As a correct replica never votes for more than one blocks with the same rank in the prepare phase within a view, it must hold that $b_1 = b_2 = b$.

**Lemma 2.** For any two prepareQCs $qc_1, qc_2$, let $b_1$ and $b_2$ be block($qc_1$) and block($qc_2$), respectively. If $b_1$ is conflicting with $b_2$, then $qc_1$.view $≠ qc_2$.view.

Proof. Assume, towards a contradiction, that $qc_1$.view $= qc_2$.view $= v$. As a valid QC consists of $2f + 1$ matching votes, at least a correct replica must have voted for both block($qc_1$) and block($qc_2$) in view $v$. We consider two cases:

1) $b_1$.height $= b_2$.height. At least one correct replica must have voted for prepareQC for both blocks with the same rank, contradicting our protocol specification.
2) $b_1.height \neq b_2.height$. We assume, w.l.o.g., $b_1.height > b_2.height$. Let $b'$ denote the block with the lowest height on the branch led by $b_1$ such that $b'.view = v$. Similarly, we define such a block $b''$ for $b_2$. Clearly, $b'.height \leq b_2.height$. Then the prepareQCs for $b'$ and $b''$ are formed in view $v$ and the parent blocks of $b'$ and $b''$ are proposed before view $v$. By Lemma 1, $b'_1 = b''_1$. Hence, $b'.height \leq b_2.height$. Let $b'_1$ be the block on the branch led by $b_1$ such that $b'_1.height = b_2.height$. Thus, $b'.height \geq b'_1.height$ and $b'_1 \neq b''_2$. So $b'_1.view = v$ and at least one correct replica has voted both $b'_1$ and $b'$ in the prepare phase in view $v$. As $b'_1$ and $b'$ have the same rank, it must hold $b'_1 = b_2$, a contradiction. □

Theorem 1. (Safety) If $b_1$ and $b_2$ are conflicting blocks, then they cannot be both committed, each by a correct replica.

Proof. Assume, on the contrary, that both $b_1$ and $b_2$ are committed, i.e., a commitQC has been formed for each block. Let $v_1$ and $v_2$ be $b_1.view$ and $b_2.view$, respectively. By Lemma 2, $v_1 \neq v_2$. W.l.o.g., we assume $v_1 < v_2$. As the commitQC for $b_1$ is formed from $2f + 1$ partial signatures, more than $f + 1$ correct replicas have received and updated their lockedQC to prepareQC for $b_1$ in view $v_1$. If any prepareQC $qc$ formed in view $v'(v_1 < v')$ is a QC for an extension of $b_1$, then the prepareQC for $b_2$ cannot be formed in view $v'$. Since correct replicas send signed responses for the commit message for $b_2$ only after receiving prepareQC for $b_2$, the commitQC for $b_2$ cannot be formed. To complete the proof, we are left to prove that for any prepareQC $qc$ formed in view $v'(v_1 < v')$, block($qc$) is an extension of $b_1$. In fact, we prove something stronger in the following lemma:

Lemma 3. If $f+1$ correct replicas have set their lockedQC to a prepareQC $qc$ in view $v$, the block of any pre-prepareQC or prepareQC formed in view $v'$ (such that $v' > v$) is an extension of block($qc$).

Proof. Let $b$ be block($qc$). Assume there exists a pre-prepareQC or prepareQC $qc'$ formed in view $v'$ for block $b'$ and $b'$ is conflicting with $b$. Let $b''$ denote the parent block of $b'$. Since $f+1$ correct replicas have set their lockedQC to $qc$ in view $v$, one of these correct replicas, say, $p_i$, must have sent a message for $qc'$ for $b'$. Let $qc_1$ be the lockedQC of $p_i$ when $p_i$ voted for $b'$ in the pre-prepare or the prepare phase. Note $qc_1$ is a prepareQC. Since $p_i$ only updates its lockedQC with a QC with a higher rank, rank($qc_1) \geq rank(qc$).

We prove the lemma by induction over the view $v'$, starting from view $v + 1$.

Base case: Suppose $qc'$ is a valid pre-prepareQC formed in view $v + 1$. It holds that $b''.view < v + 1$, $qc'.view = v$, and $qc_1.height \geq qc'.height$. From Lemma 2, block($qc_1$) does not conflict with $b$, so block($qc_1$) is $b$ or an extension of $b$. We consider two cases: $b'$ is a normal block; $b'$ is a virtual block.

1) If $b'$ is a normal block, then $b'.justify$ contains a QC $qc''$ for $b''$. Since $p_i$ has voted for $b'$, one of the following two conditions must be satisfied:

- $qc'.view < v + 1$ and rank($qc''$) $\geq$ rank($qc_1$) (Case R1). In this situation, type($qc''$) = PREPARE, $qc''$.view = $v$, and $qc''$.height $\geq$ $qc_1$.height. From Lemma 2, $b''$ is $b$ or an extension of $b$. Thus, $b'$ must be an extension of $b$.

- $b'' = block(qc_1)$ (Case R3). In this situation, $b''$ must be an extension of $b$, as block($qc_1$) is $b$ or an extension of $b$.

Either way, $b'$ cannot conflict with $b$, a contradiction.

2) If $b'$ is a virtual block, then there exists a prepareQC $vc'$ for $b''$, such that $vc'.view < v + 1$ and rank($vc'$) $\geq$ rank($qc_1$). By Lemma 2, $b''$ must be $b$ or an extension of $b$, a contradiction.

Suppose $qc''$ a prepareQC formed in view $v + 1$. Let $qc''$ denote the prepareQC with the lowest height formed in view $v + 1$. Then when block($qc''$) is broadcast in PREPARE message, a valid pre-prepareQC is provided. Therefore, block($qc''$) is an extension of $b$. By Lemma 2, block($qc''$) cannot be conflicting with block($qc''$). Since $qc''$.height $\geq$ $qc''$.height, $b'$ must be an extension of $b$, a contradiction.

Inductive case: Assume this property holds for view $v'$ from $v$ to $v + k - 1$ for some $k \geq 1$. We prove that it holds for $v' = v + k$. Suppose $qc'$ is a valid pre-prepareQC formed in view $v + k$. According to the inductive hypothesis, block($qc_1$) is $b$ or an extension of $b$. We distinguish two cases:

1) $b'$ is a normal block, then $b'.justify$ contains a QC $qc''$ for $b''$. Since $p_i$ has voted for $b'$, one of the following two conditions must be satisfied:

- $qc''.view < v + k$ and rank($qc''$) $\geq$ rank($qc_1$) (Case R1)
- $b'' = block(qc_1)$ (Case R3). In this situation, $b''$ must be an extension of $b$, because block($qc_1$) is $b$ or extension of $b$.

Either way, $b''$ cannot conflict with $b$, a contradiction.

2) If $b'$ is a virtual block, then there exists a prepareQC $vc'$ for $b''$ such that $vc'.view < v + k$ and rank($vc'$) $\geq$ rank($qc_1$). Again, by Lemma 2 and the inductive hypothesis, $b''$ must be an extension of $b$, a contradiction.

Now we assume $qc'$ is a prepareQC formed in view $v + k$. Let $qc''$ be the prepareQC with the lowest height formed in view $v + k$. Then a valid pre-prepareQC for block($qc''$) must have been formed. Thus, block($qc''$) is an extension of $b$. From Lemma 2, $b''$ must be an extension of $b$, a contradiction. □

Lemma 4. In view change, one of the following must hold for $highQC_v$ received by the leader: 1) $highQC_v$ is a pre-prepareQC; 2) $highQC_v$ contains one pre-prepareQC; 3) $highQC_v$ contains two pre-prepareQCs with the same rank.

Proof. First note that $highQC_v$ contains at least one QC that is either a pre-prepareQC or a prepareQC. If a prepareQC $qc$ is included in $highQC_v$, then due to Lemma 2, $highQC_v$ contains $qc$ only. Hence, $highQC_v$ is a prepareQC (Case 1).

If no prepareQC is included in $highQC_v$, then $highQC_v$ contains one pre-prepareQC (Case 2), or more pre-prepareQCs, in which case these pre-prepareQCs are QCs formed in the same view. Since correct replicas vote for at most two blocks in the pre-prepare phase, at most two pre-prepareQCs with the same rank can be formed within a view. In this case, $highQC_v$ contains at most two pre-prepareQCs of the same rank (Case 3). □
Lemma 5. After GST, there exists a bounded time period $T$ such that if all correct replicas remain in view $v$ during $T$ and the leader for view $v$ is correct, then a valid pre-prepareQC can be formed.

Proof. In view change, the leader $l_v$ collects $n - f$ VIEW-CHANGE messages and calculates its highQC to propose a new block. By Lemma 4, one of the three cases must apply. Suppose among all correct replicas, the lockedQC with the highest rank is a prepareQC $qc$ for $b$. Let $b'$ denote the parent block of $b$, $v_b$ denote $b$.view. We distinguish two cases:

1) If $b'.view < v_b$, then by Lemma 1, no prepareQC with a lower height than $b$ can be formed in view $v_b$. The lockedQC of any correct replica is either $qc$ or a prepareQC formed before $v_b$. As prepareQC for $b$ is formed in view $v_b$, at least $f + 1$ correct replicas have set their highQC to prepareQC $qc'$ for $b$. Thus, $M_v$ contains at least one VIEW-CHANGE message from these replicas. Since correct replicas update their highQC with prepareQC or prepareQC with a higher rank, the rank of QC(s) in highQC $v$ is no less than rank($qc'$).

If the rank of QC(s) in highQC $v$ is equal to rank($qc'$), then highQC $v$ contains $qc'$. Then the block proposed by $l_v$ extending $b$ will be voted by all correct replicas for pre-prepareQC in view $v$, as either Case R1 or R3 is satisfied.

If the rank of QC(s) in highQC $v$ is higher than rank($qc'$), then for any QC $qc_b$ in highQC $v$, $qc_b.view > v_b$ or $(qc_b.view = v_b$ and type($qc_b$) = PREPARE). Either way, by Lemma 1, rank($qc_b$) $\geq$ rank($qc$). Then the block(s) proposed by $l_v$ in view $v$ will be voted by all correct replicas for pre-prepareQC, because Case R1 is satisfied.

2) If $b'.view = v_b$, then $b$.justify is a prepareQC $qc'$ for $b'$. At least $f + 1$ correct replicas have set their highQC to $qc'$. Thus, $M_v$ contains at least one VIEW-CHANGE message from these replicas. The rank of QC(s) in highQC $v$ is no less than rank($qc'$). If the rank of QC(s) in highQC $v$ is equal to rank($qc'$), then due to Lemma 2, highQC $v$ is $qc'$. In this case, $l_v$ proposes two blocks, one normal block $b_1$ with height $b'.height + 1$ and a virtual block $b_2$ with height $b'.height + 2$. At least $b_2$ can be voted by all correct replicas to form a prepareQC $qc_v$, since Case R1 or R2 is satisfied. If the prepareQC $vc$ for $b'$ is received by $l_v$, $l_v$ updates its highQC to $(qc_v, vc)$ and $qc_v$ is a valid pre-prepareQC. Otherwise a valid prepareQC for $b_1$ can be formed.

If the rank of QC(s) in highQC $v$ is higher than rank($qc'$), we know the rank of QC(s) in highQC $v$ is no less than rank($qc$). The block(s) proposed by $l_v$ will be voted by all correct replicas for pre-prepareQC, as Case R1 is satisfied.

Theorem 2. (Liveness) After GST, there exists a bounded time period $T_f$ such that if all correct replicas remain in view $v$ during $T_f$ and the leader for view $v$ is correct, then an decision can be reached.

Proof. By Lemma 5, a valid pre-prepareQC $qc$ can be formed in the new view if the leader is correct. Then the block of $qc$ can be accepted by all correct replicas in the prepare phase since the rank of $qc$ is higher than the lockedQC for any correct replicas. Replicas can then resume the normal case operation and a decision can be reached.

VI. IMPLEMENTATION AND EVALUATION

Overview. We implement Marlin and HotStuff in Go using around 7,000 LOC, including 1,500 LOC for evaluation. We implement the chaining (pipelining) mode for both Marlin and HotStuff. We deploy the protocols in a cluster with 40 servers. Each server has a 16-core 2.3GHz CPU, 128 GB RAM, 1000 MB NIC. We use $f$ to represent the network size, where we use $3f + 1$ replicas in each experiment. The network bandwidth is 200 Mbps. We injected 40ms network latency for all experiments carried. Except for the experiment for no-op requests (containing digital signatures but no operations), all transactions and reply messages are of size 150 bytes. We use LevelDB as the underlying database. The frequency of garbage collection (checkpointing) is set to every 5000 blocks. We use ECDSA as the underlying signature. Our main finding is unlike other HotStuff variants that are at least sometimes less efficient than HotStuff (as reported in [30, 31]), Marlin consistently outperforms HotStuff.

Throughput vs. latency. We first assess the throughput vs. latency of the in failure-free scenarios for both Marlin and HotStuff. We report throughput vs. latency for $f = 1$ to $f = 30$ in Figure 10a-10f. As shown in the figures, by reducing the number of phases of HotStuff from three to two, the throughput of Marlin is 4.47%-34.4% higher than that of HotStuff. In particular, when $f = 1$, Marlin achieves peak throughput of 101 ktx/sec, 27.2% higher than that of HotStuff.

Note that our implementation appears to have lower performance than those in prior works [2, 30, 31, 52]. This reason is that our implementation writes data into the database rather than into memory and we run checkpointing in the backend. Thus, our experiments are more realistic than prior ones.

Scalability. We report the peak throughput of Marlin and HotStuff for $f = 1$ to $f = 10$ in Figure 10g. The peak throughput of Marlin is 11.56%-34.4% higher than that of HotStuff. When $f$ grows, the throughput of Marlin degrades in a way much like HotStuff. In our case, when $f$ is greater than 5, the performance downgrades significantly. However, even when $f = 10$, the throughput of Marlin can still be as large as 23.82 ktx/s.

We also conduct the same experiments for no-op requests. Due to space limitation, we only report the performance for $f = 1$, $f = 2$, and $f = 5$ in Figure 10h. The performance of these no-op experiments for both Marlin and HotStuff is consistently higher than the one for experiments with larger requests and replies (150 bytes). For instance, when $f = 1$, the peak throughput for no-op requests is 16.7% higher than that with the large request size. When $f$ increases, the performance does not downgrade as much as that for large request sizes. For instance, when $f = 5$, the peak throughput of Marlin is still 101 ktx/s, almost twice higher than the experiment using 150-byte requests. In other words, the protocols are more scalable for smaller requests and replies.
In our experiments, we let leader mode, we follow HotStuff implementation [1] (setting of the rotating leader mode under failures. By the rotating we assess the performance Performance under failures. in contrast, is similar to HotStuff for both $f = 1$ and $f = 10$. The results show that the view change protocol of Marlin is at least as efficient as HotStuff. In practice, one could anticipate the average latency of Marlin would be somewhere between the happy path latency and the unhappy path latency.

Performance under failures. We assess the performance of the rotating leader mode under failures. By the rotating leader mode, we follow HotStuff implementation [1] (setting up a timer) and Spinning [49] to rotate leaders periodically. In our experiments, we let $f = 3$ and crash 1 or 3 replicas at the beginning of the experiments. We set up the timer for the rotating leader to 1s. We report the performance of the failure-free case, the case under 1 failure, and the case under 3 failures in Figure 10j. Both Marlin and HotStuff suffer from performance degradation under failures. For the 1 failure scenario, the performance of Marlin and HotStuff is 24.5% and 26.8% lower than that in the failure-free case, respectively. When there are 3 failures, the performance of Marlin and HotStuff is 36.11% and 38.66% lower, respectively. The results are expected, as no requests can be proposed or committed when a faulty replica is a leader. In all the cases, Marlin consistently outperforms HotStuff. For instance, when there are 3 failures, the throughput of Marlin is 34.8% higher than that of HotStuff.

**Performance of view changes.** We evaluate the performance of view changes. We compute the view change latency from the point when a replica starts the view change to the point when the first block is committed after the view change. We first submit a few client requests and then crash the leader to assess the performance of view change. For Marlin, we force our code to execute both happy and unhappy paths to fully understand the performance. As shown in Figure 10i, the latency for HotStuff is 182 ms when $f = 1$ and 384 ms when $f = 10$. Meanwhile, in the happy path, the latency for Marlin is 123 ms for $f = 1$ and 229 ms for $f = 10$, being about 30% to 40% lower than HotStuff. The latency for the unhappy path, in contrast, is similar to HotStuff for both $f = 1$ and $f = 10$. The results show that the view change protocol of Marlin is at least as efficient as HotStuff. In practice, one could anticipate the average latency of Marlin would be somewhere between the happy path latency and the unhappy path latency.

**VII. Conclusion**

This paper introduces Marlin, a novel BFT protocol that commits operations in two phases and has a linear authenticator communication. We prove the correctness of Marlin and provide an efficient implementation for Marlin and HotStuff. Via extensive evaluation, we show Marlin outperforms HotStuff in various scenarios.

**Acknowledgement**

We thank DSN reviewers and our shepherd Vivien Quémé for their insightful comments.
