

Dashing and Star: Byzantine Fault Tolerance with Weak Certificates

Sisi Duan
Tsinghua University
duansisi@tsinghua.edu.cn

Baohan Huang
Shandong University
huangbaohan@mail.sdu.edu.cn

Haibin Zhang
Beijing Institute of Technology
haibin@bit.edu.cn

Changchun Mu
Digital Currency Institute, the
People’s Bank of China
mchangchun@pbc.gov.cn

Xiao Sui
Shandong University
suixiao@mail.sdu.edu.cn

Gang Di
Tsinghua University
Digital Currency Institute, the
People’s Bank of China
digang@pbc.dci.cn

Xiaoyun Wang
Tsinghua University
xiaoyunwang@tsinghua.edu.cn

Abstract

State-of-the-art Byzantine fault-tolerant (BFT) protocols assuming partial synchrony such as SBFT and HotStuff use *regular certificates* obtained from $2f + 1$ (partial) signatures. We show that one can use *weak certificates* obtained from only $f + 1$ signatures to *assist* in designing more robust and more efficient BFT protocols. We design and implement two BFT systems: Dashing (a family of two HotStuff-style BFT protocols) and Star (a parallel BFT framework).

We first present Dashing that targets both efficiency and robustness using weak certificates. Dashing1 is also network-adaptive in the sense that it can leverage network connection discrepancy to improve performance. We show that Dashing1 outperforms HotStuff in various failure-free and failure scenarios. We further show in Dashing2 how to further enable a *one-phase* fast path by using *strong certificates* obtained from $3f + 1$ signatures, a challenging task we tackled.

We then leverage weak certificates to build Star, a highly scalable BFT framework that delivers transactions from $n - f$ replicas. Star compares favorably with existing protocols in terms of censorship resistance, communication complexity, pipelining, state transfer, performance and scalability, and/or robustness under failures.

We demonstrate that Dashing achieves 47%-107% higher peak throughput than HotStuff for experiments on Amazon EC2. Meanwhile, unlike all known BFT protocols whose performance degrades as f grows large, the peak throughput of Star keeps increasing as f grows. When deployed in a WAN with 91 replicas across five continents, Star achieves an impressive throughput of 256 ktx/sec, 35.9x that of HotStuff, 23.9x that of Dashing1, and 2.38x that of Narwhal.

1 Introduction

Byzantine fault-tolerant state machine replication (BFT) is known as the core building block for permissioned blockchains

[5, 7, 16, 33, 35, 46, 50]. This paper focuses on highly efficient, partially synchronous BFT protocols [13, 20]. Almost universally, these protocols rely critically on *regular (quorum) certificates* which, roughly speaking, are sets with at least $2f + 1$ messages from different replicas. Recent protocols such as SBFT [26] and HotStuff [52] require using (threshold) signatures for regular certificates as transferable proofs.

This paper demonstrates that one can build BFT systems that outperform existing ones—in one way or another—by using *weak certificates* with at least $f + 1$ signatures from different replicas.

Intuitively, weak certificates may lead to more efficient BFT protocols, because replicas only need to wait for signatures from $f + 1$ replicas and combine only $f + 1$ signature shares. Indeed, as shown in prior works (e.g., [19]), Byzantine agreement protocols with the $f + 1$ threshold can be (much) more efficient than their counterparts with the $2f + 1$ threshold. *This paper explores novel usages of weak certificates much beyond this intuition.*

Table 1 summarizes our protocols using weak certificates. The Dashing protocols (Dashing1 and Dashing2) are new BFT protocols in the HotStuff family and gain in efficiency during failure-free cases and robustness under unexpected network interruptions. Star is a new asynchronous BFT framework targeting scalability.

1.1 Dashing: Gaining in Efficiency, Network Adaptivity, and Robustness

In Dashing, we challenge the conventional wisdom and offer new insights into the design of BFT protocols.

- **Using weak certificates.** It is well-known that BFT protocols need to use regular certificates to ensure liveness and safety. So far, weak certificates do not appear to be helpful in building faster BFT protocols. Our first goal is to challenge the intuition and provide a way to exploit weak certificates to assist in the BFT design.

| protocols | section | QC type used | features | authenticator | communication |
|-----------|----------|---------------|--|---------------|-------------------------|
| Dashing1 | Sec. 3.4 | wQC; rQC | network-adaptive; more robust and efficient | $O(n)$ | $O(Ln + \lambda n)$ |
| Dashing2 | Sec. 3.5 | wQC; rQC; sQC | targeting low latency; one-phase fast path | $O(n)/O(n^2)$ | $O(Ln + \lambda n^2)$ |
| Star | Sec. 4 | wQC; rQC | 1. pipelined transmission; 2. weak certificates for efficiency; 3. effective blockchain quality; 4. efficient state transfer; 5. lower communication | $O(n^2)$ | $O(Ln^2 + \lambda n^2)$ |

Table 1. Our protocols. L is the proposal size for each replica and λ is the security parameter. As Star allows replicas to process $n - f$ transactions in parallel, one cannot simply say that the Dashing protocols have lower communication complexity than Star.

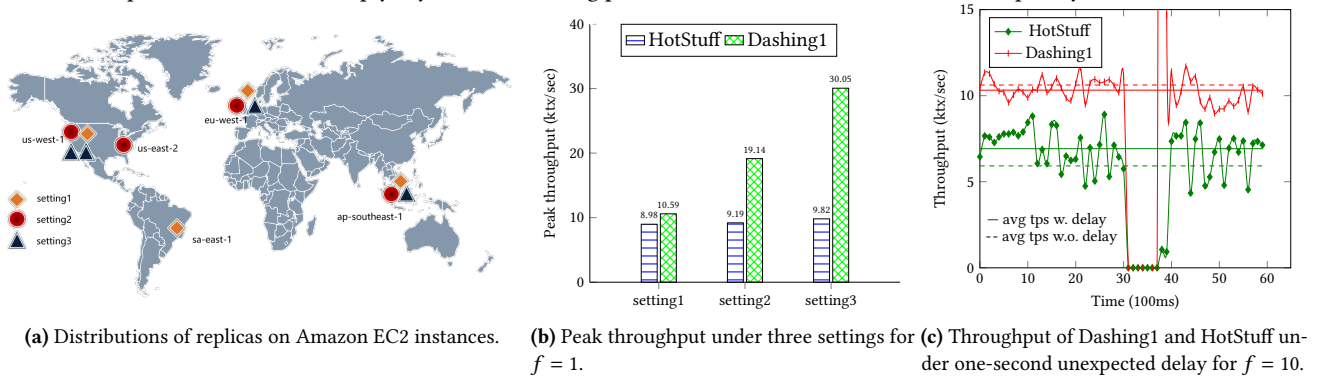


Figure 1. Throughput of HotStuff and Dashing1 in three different settings on Amazon EC2.

• **Leveraging network connection discrepancy.** When designing and evaluating a partially synchronous BFT, we implicitly assume the simplistic network configuration, where replicas communicate with each other with about the same latency (either all in LANs or WANs). But in practice, the latency discrepancy among different replicas naturally exists. A realistic scenario is that some replicas (say, 1/3 of the replicas) naturally have better connections than the rest of them. We find the fact is overlooked by existing BFT protocols. We experimentally show in Fig. 1 that HotStuff does not exhibit visible performance differences even if we place some replicas in the same region. The result is somewhat expected: intuitively, the safety of BFT depends on the BFT network overall, so the performance of BFT should depend on the BFT network overall. Again, we challenge this intuition, showing that BFT can benefit from network connection discrepancies.

• **Useful work during asynchrony.** Partially synchronous BFT protocols cannot make progress during asynchrony. Existing partially synchronous BFT protocols would simply wait until the network becomes synchronous (before view change/leader election occurs) or loop on view changes until a correct leader is selected—in either case, no meaningful progress can be made. The situation is only exacerbated, if the network is intermittently synchronous or adaptively manipulated [42]. Naturally, it seems that there is nothing we can do about the situation: existing partially synchronous BFT protocols are deterministic and subject to the celebrated FLP impossibility result [21]. We take a fresh look at the problem: while one indeed cannot make progress during asynchrony, we do not waste our computation and network bandwidth during asynchrony. The idea is that we perform

"useful" operations such that once the network becomes synchronous, we can efficiently commit a large number of cumulative transactions—the longer the asynchrony, the more transactions committed—in some sense, the "best" that one could anticipate.

Dashing1. One crucial idea in Dashing1 is that we attempt to use weak certificates instead of regular certificates *as much as possible*—during the normal case, during transient failures or network interruptions, during unresponsive replicas (e.g., crashes, slow replicas), and during view changes. Transforming the idea into a fully secure BFT protocol, however, is tricky: we have tackled subtle safety and liveness challenges within a view and across views due to the usage of weak certificates. Correspondingly, Dashing1 gains in improved efficiency and robustness in various scenarios, including during normal cases and across views, and in the presence of transient network interruptions, network connection discrepancies, or unresponsive failures.

As shown in Fig. 1a, we deploy HotStuff and Dashing1 on Amazon EC2 (for $n = 4$) in three different settings: in setting 1, the four replicas are distributed over four continents; in setting 2 and setting 3, we place two replicas in closer locations. In all three settings (Fig. 1b), we find Dashing1 consistently outperforms HotStuff; in setting 1 and setting 2, Dashing1 achieves about 2x and 3x the throughput of HotStuff, respectively. The experiments show that Dashing1 achieves improved performance in the normal case and in the presence of (natural) network connection discrepancies.

As another example (Fig. 1c), we run an experiment for Dashing1 and HotStuff with 1,200 clients for a duration of six seconds in a WAN setting with 31 replicas. In the middle of the experiments, we inject a one-second network delay using

the *qdisc* traffic control command. While neither HotStuff nor Dashing1 can make progress during the network delay, the throughput of Dashing1 reaches roughly 10x that of HotStuff when the network recovers. The average throughput of Dashing1 is 79.3% and 49.1% higher than that of HotStuff with the unexpected network delay (dashed line) and without the delay (solid line), respectively. Moreover, Dashing1 achieves roughly the same average throughput as that without the delay, while we witness a more visible decrease in throughput for HotStuff.

We also show in the paper that Dashing1 enjoys better robustness and efficiency in various other scenarios such as leader failures or backup failures.

Dashing2. We show how to enable a one-phase fast path by leveraging *strong certificates* from $3f + 1$ signatures in our BFT protocols. We demonstrate that such a task is technically challenging—being more subtle than that in SBFT [26]—and offer a secure and efficient solution.

1.2 Star: Gaining in Efficiency and Scalability

We use weak certificates to help build Star, a highly scalable BFT framework that delivers transactions from $n - f$ replicas using only a single consensus instance. As depicted in Fig. 2, Star completely separates bulk data transmission from consensus such that these two processes can be run independently, an idea originally from [1]. Star has five distinctive features compared to prior works: 1) the data transmission process can be effectively *pipelined* to gain in efficiency; 2) Star uses weak certificates for the data transmission process to further improve performance; 3) unlike prior works, the transmission process and the consensus process are implicitly "correlated" with epoch numbers, and the consensus process only handles messages transmitted in the same epoch, which helps achieve effective censorship resilience and *blockchain quality* (at least 1/2 of the total transactions contained in a committed block in an epoch are from correct replicas); 4) Star admits a more efficient ($O(1)$ time) state transfer mechanism outpacing existing ones; and 5) Star achieves lower communication complexity than existing protocols.

All the features add up to a highly scalable and robust BFT framework. Simply using PBFT [14] in our underlying consensus layer, Star is the first conventional BFT protocol whose throughput strictly keeps increasing as n grows. As illustrated in Fig. 3, when deploying Star, HotStuff, Narwhal [1] (the state-of-the-art protocol¹) in a WAN with 91 replicas across five continents, Star achieves a throughput of 256 ktx/sec, 35.9x that of HotStuff and 2.38x that of Narwhal. **Comparison with existing protocols.** Besides the performance and scalability benefits, Star compares favorably

¹As claimed by authors in Bullshark [25], Bullshark and Narwhal share almost identical throughput in normal cases, and BullShark offers almost 2x the throughput of Mir-BFT at the same latency.

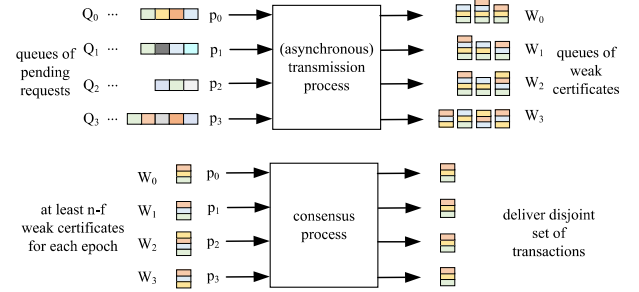


Figure 2. The Star BFT framework. Star consists of an asynchronous transmission process (that takes as input queues of pending transactions and outputs queues of weak certificates) and a consensus process (that takes at input $n - f$ weak certificates and outputs a union of transactions corresponding to the weak certificates delivered). Two processes are run independently but are implicitly correlated with an increasing epoch number.

with existing protocols, such as Narwhal [1], Bullshark [25], ISS [48], Mir-BFT [47], and Dumbo-NG [22], in terms of liveness, message complexity, pipelined transmission, state transfer, and/or performance under failures.

DAG-based BFT protocols have liveness issues. DAG Rider [32] require unbounded memory for liveness, while Bullshark [25] and Tusk [1] achieve non-standard and weak liveness.

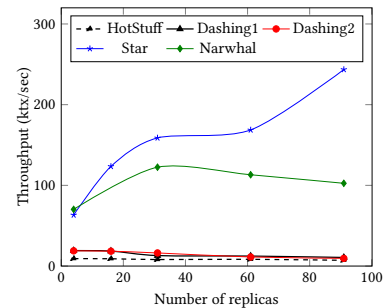


Figure 3. Throughput of the protocols in WANs as f grows.

Concurrent to our work, Gao, Lu, Lu, Tang, Xu, and Zhang proposed Dumbo-NG [22], an asynchronous BFT protocol. Indeed, while we instantiate Star using a partially synchronous BFT protocol, Star can be an asynchronous BFT protocol if the underlying BFT is asynchronous. Moreover, both Dumbo-NG and Star separate message transmission from agreement. However, Dumbo-NG does not use any of the other three techniques we used to improve efficiency or blockchain quality: 1) weak certificates for better efficiency; 2) associating transmission layer and consensus layer with epoch numbers for better blockchain quality; 3) a more efficient state transfer mechanism. In particular, without associating transmission layer with consensus layer, a specific transaction could be significantly delayed or censored due to faster commitments of transactions initiated by faulty replicas. Indeed, faulty replicas can form a long chain consisting

of an unbounded number of certificates. In this way, a specific transaction may have the opportunity to be processed only after all associated transactions from faulty replicas have been committed, losing constant commit time; moreover, the fraction of transactions from correct replicas in a block may be made arbitrarily small. Last, to achieve liveness, Dumbo-NG also requires unbounded storage (a fact confirmed by Dumbo-NG authors).

Different from ISS [48] (and its predecessor Mir-BFT [47]) that requires running n parallel consensus protocols for each epoch, Star only needs to run a single consensus protocol for each epoch. ISS relies on Byzantine failure detector to ensure safety and liveness, and replicas have to wait for the slowest consensus instance to terminate (possibly with view changes or until timers run out) before they can process transactions; in contrast, Star can process transactions once the single consensus instance completes. Also, Star achieves $O(n^2)$ messages, which is in contrast to ISS that incurs at least $O(n^3)$ messages. Last, during crash failures, the throughput of ISS and Mir-BFT may drop to 0 for a long period of time; it needs to run a reconfiguration mechanism to exclude faulty replicas [47, 48].

1.3 Summary of Contributions

We summarize our contributions in the following.

- We design a family of Dashing protocols—Dashing1 and Dashing2—using weak certificates. In particular, Dashing1 gains in improved efficiency and robustness in both failure and failure-free scenarios and in normal cases and across views; unlike prior partially synchronous protocols, Dashing1 excels in performance in the presence of transient network interruptions and network connection discrepancies. Dashing2 enables a one-phase fast path for Dashing1 and offers improved latency.
- We provide a novel (asynchronous) BFT framework (Star) allowing one to process transactions in parallel using only one BFT instance and $O(n^2)$ messages. Star separates a pipelined message transmission layer from the consensus layer, yet associates the layers using an increasing epoch number. Doing so allows us to achieve a strong blockchain quality property. Additionally, Star enables an $O(1)$ time state transfer (recovery) mechanism that is much more efficient than existing ones.
- We implement the BFT protocols (the two Dashing protocols and a Star instantiation). We performed extensive evaluations of the protocols, showing that our protocols outperform existing protocols in various metrics.

2 System Model

BFT. This paper studies Byzantine fault-tolerant state machine replication (BFT) protocols. In a BFT protocol, clients *submit* transactions (requests) and replicas *deliver* them. The client obtains a final response to the submitted transaction

from the replica responses. In a BFT system with n replicas, it tolerates $f \leq \lfloor \frac{n-1}{3} \rfloor$ Byzantine failures. The correctness of a BFT protocol is specified as follows:

- **Safety:** If a correct replica *delivers* a transaction tx before *delivering* tx' , then no correct replica *delivers* a transaction tx' without first *delivering* tx .
- **Liveness:** If a transaction tx is *submitted* to all correct replicas, then all correct replicas eventually *deliver* tx .

Liveness is alternatively called "censorship resilience" (a blockchain terminology). We use them interchangeably.

We also need an equivalent primitive, atomic broadcast, as a building block. Atomic broadcast is only syntactically different from BFT. In atomic broadcast, a replica *a-broadcasts* messages and all replicas *a-deliver* messages.

- **Safety:** If a correct replica *a-delivers* a message m before *delivering* m' , then no correct replica *a-delivers* a message m' without first *a-delivering* m .
- **Liveness:** If a correct replica *a-broadcasts* a message m , then all correct replicas eventually *a-deliver* m .

Note that when describing atomic broadcast, we restrict the API of atomic broadcast in the sense that only a single replica *a-broadcasts* a message. One can alternatively allow all replicas to *a-broadcast* transactions (which is the case for completely asynchronous protocols).

This paper mainly considers the partially synchronous model [20], where there exists an unknown global stabilization time (GST) such that after GST, messages sent between two correct replicas arrive within a fixed delay. One of our protocols (Star) works in purely asynchronous environments if the underlying atomic broadcast is asynchronous.

(Best-effort) broadcast. We use the term "broadcast" to represent the event that a replica sends a message to all replicas in a system.

Cryptographic building blocks. We define a (t, n) threshold signature scheme with the following algorithms ($tgen$, $tsgn$, $tcombine$, $tverify$). $tgen$ outputs a threshold signature public key and a vector of n private keys. A signature signing algorithm $tsgn$ takes as input a message m and a private key sk_i and outputs a partial signature σ_i . A combining algorithm $tcombine$ takes as input pk , a message m , and a set of t valid partial signatures, and outputs a signature σ . A signature verification algorithm $tverify$ takes as input pk , a message m , and a signature σ , and outputs a single bit. We require the robustness and unforgeability properties for threshold signatures. When describing the algorithms, we leave the verification of partial signatures and threshold signatures implicit. Dedicated threshold signatures can be realized using pairings [10, 11]. One can also use a group of signatures to build a (t, n) threshold signature for efficiency, as used in various libraries such as HotStuff [2, 52], Jolteon and Ditto [23], and Wendy [24]. The approach is also preferred for our protocols, as many of our protocols have more

than one threshold. (Otherwise, one should use different threshold signatures for different thresholds.)

We use a collision-resistant hash function *hash* mapping a message of arbitrary length to a fixed-length output.

Byzantine quorums and quorum certificates. We assume $n \geq 3f + 1$ for our protocols. For simplicity, we let $n = 3f + 1$. A Byzantine quorum consists of $\lceil \frac{n+f+1}{2} \rceil$ replicas, or simply $2f + 1$ if $n = 3f + 1$. We call it a *regular quorum*.

Slightly abusing notation, we additionally define two different types of quorums: a *weak quorum* consisting of $f + 1$ replicas and a *strong quorum* consisting of $n = 3f + 1$ replicas. A message with signatures signed by a weak quorum, a regular quorum, and a strong quorum is called a *weak (quorum) certificate* (wQC), a *regular (quorum) certificate* (rQC), and a *strong (quorum) certificate* (sQC), respectively. A certificate can be a threshold signature with a threshold t or a set of t digital signatures.

3 The Family of Dashing Protocols

3.1 Overview of (Chained) HotStuff

HotStuff describes the syntax of leader-based BFT replication using the language of trees over blocks for leader-based protocols. Here we use a slightly more general notation, where multiple blocks, rather than just one block, may be delivered within a view until view change occurs.

Each replica stores a tree of blocks. A block b contains a parent link pl , a batch of transactions, and their metadata. A parent link for b is a hash 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 (i.e., the *genesis* block). The *height* for b is the number of blocks on the branch led by b .

Each time a monotonically growing branch becomes committed, a block extends the branch led by its parent block. A block b' is an extension of a block b , if b is on the branch led by b' . Two branches are conflicting, if neither is an extension of the other. Two blocks are conflicting, if the branches led by the blocks are conflicting. A safe BFT ensures that no two correct replicas commit two conflicting blocks.

HotStuff uses three phases (*prepare phase*, *precommit phase*, and *commit phase*) to deliver a block. In the prepare phase, the leader broadcasts a proposal (a block) b to all replicas and waits for signed responses (also called votes) from a quorum of $n - f$ replicas to form a threshold signature as a quorum certificate (*prepareQC*). In the following precommit phase, the leader broadcasts *prepareQC* and waits for responses to form *precommitQC*. Similarly, in the commit phase, the leader broadcasts *precommitQC*, and waits to form *commitQC* and then broadcasts it. Upon receiving the *precommitQC*, a replica becomes *locked* on b . Upon receiving the *commitQC*, a replica delivers b .

During view changes, each replica sends the leader its latest *prepareQC*. Upon receiving a quorum of $n - f$ such

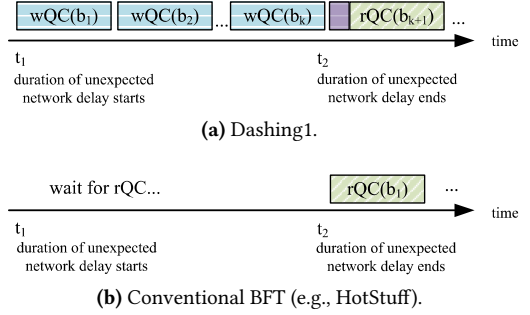


Figure 4. The way how Dashing1 and a regular BFT handle unexpected network delays, respectively.

messages, the leader selects the QC of largest height and extends the block for the QC using a new proposal.

Throughout the paper, we use the chained version for HotStuff and the Dashing protocols, where phases are overlapped and pipelined.

3.2 Overview of Dashing1

Dashing1 in a nutshell. In Dashing1, we use weak certificates (signatures from $f + 1$ replicas) to improve both efficiency and robustness. The core idea is to use weak certificates *as much as possible*, during normal cases and across views, and in the presence of transient network interruptions and network connection discrepancies. In any of the above cases, we allow replicas to "proceed" with weak certificates.

As an example, consider a system with seven replicas, p_1 to p_7 . The leader p_1 can only receive messages from p_2 and p_3 , but not other replicas. As long as the network interruption persists, replicas in existing BFT protocols cannot make meaningful progress. For partially synchronous BFT protocols, the leader has to wait until a regular certificate is formed, or until view change occurs. In contrast, Dashing1 allows replicas to make meaningful progress and accumulate proposals under unexpected delays.

Fig. 4a describes the way how Dashing1 and a regular BFT protocol (e.g., HotStuff) handle unexpected network delays. For both protocols, starting from time t_1 , the leader could not form an rQC until t_2 when the network becomes synchronous again. During the network delay, the leader in the regular BFT protocol proposes block b_1 and waits for its rQC.

In contrast, for the case of Dashing1, the leader can form a sequence of wQCs for blocks b_1, \dots, b_k . Then when the network becomes synchronous, replicas that fail to receive messages from the leader will catch up with the leader. This catching up phase, as we find, completes in a very short period of time, in contrast to proposing blocks and collecting wQCs. Then replicas resume their normal-case operation and rQCs can be formed. After three rQCs for block b_{k+1} are formed, blocks b_1, \dots, b_{k+1} are committed simultaneously. Namely, during expected network delays or interruptions, we do not waste our computation and network bandwidth; once

the network becomes synchronous, we can commit all the cumulative transactions simultaneously and instantaneously.

Note that if $t_2 - t_1$ is larger than the view change timer, view change will be triggered. Even during view change, the design of Dashing1 allows the new leader (if correct) to create a new proposal based on wQCs from the prior view. Therefore, the view change protocol in Dashing1 can also benefit from our design.

With our design, Dashing1 can naturally leverage network connection discrepancy for adaptive performance. The performance of Dashing1 depends on the group of fast replicas (1/3 of total replicas) rather than the Byzantine quorum of replicas (the overall network condition). Meanwhile, we can carefully set up timers so that our system can benefit from wQCs even in the normal case.

Challenges and our design. Transforming the idea into a fully secure BFT protocol, however, is non-trivial. First, a faulty leader may easily create forks and generate up to $2f + 1$ conflicting weak certificates. To prevent the forks from growing exponentially, we can ask each correct replica to vote for at most one block at each height.

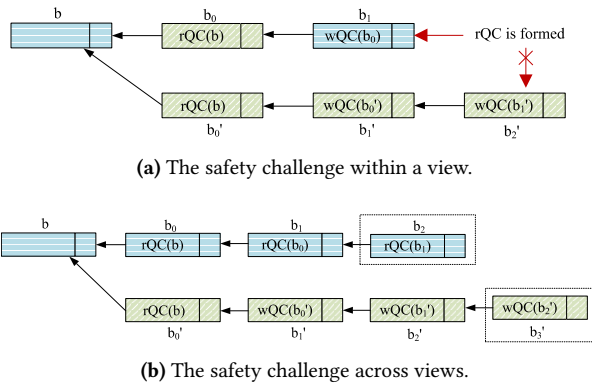


Figure 5. Challenges of building BFT from weak certificates. Blue boxes with horizontal lines and green boxes with diagonal lines represent blocks such that regular certificates and weak certificates are formed, respectively. (The figures are best viewed in color.)

Second, we need to ensure that the protocol achieves safety within a view even if wQCs are used. Namely, if forks are formed, a regular certificate can only be formed for at most one of the branches. As shown in Fig. 5a, b_0 and b'_0 are conflicting blocks and wQCs are formed for both of them. In addition, b_1 extends b_0 and b'_1 extends b'_0 . Then a regular certificate is formed for b_1 . While a wQC can still be formed for b'_1 and its descendant blocks, we need to ensure that a regular certificate will never be formed for any of them. We solve the problem by enforcing a constraint: if a replica receives a proposal for a block (e.g., b'_2) that extends a block with a wQC (i.e., b'_1), the replica votes for block b'_2 if and only if it has previously voted for the parent block b'_1 . As $2f + 1$

replicas have already voted for b_1 , it is impossible that $2f + 1$ replicas will vote for b'_2 .

Third, the protocol should achieve safety across views. During view changes, we ask each replica to send its *highest* wQC to the new leader and the new leader can select a branch led by a wQC when creating a new proposal. However, letting a new leader select the highest wQC it receives may lead to a subtle safety problem: the new leader may not choose the *right* wQC. As shown in Fig. 5b, rQCs are formed for b_0 , b_1 , and b_2 , while wQCs for b'_0 , b'_1 , b'_2 , and b'_3 are formed too (a "fork"). Note that an rQC for b_2 is also the *commitQC* for b_0 . If a view change occurs and the leader selects the highest weak certificate (a wQC for b'_3), a *conflicting proposal* with the committed block b_0 will be proposed.

To address this issue, for any block b , we additionally define a *stable block* as the highest block with an rQC on the branch led by b . After a new leader collects the certificates from $2f + 1$ replicas, it will select a *safe block* to extend based on the highest rQC and the wQC with the highest stable block. In this example, as the stable block of b'_3 is b and b is lower than b_2 , the leader will create a proposal extending b_2 .

One more (liveness) challenge is about timers. In Dashing1, besides the regular view change timer Δ_1 , the leader additionally maintains a timer Δ_2 . After forming a wQC with $f + 1$ matching votes, the leader starts a timer Δ_2 . The leader can propose a new block if either of the two conditions is met: 1) Δ_2 expires (and it has not formed an rQC); 2) it forms an rQC. We need to be careful about Δ_2 . According to our experiments, even if Δ_2 is set as a relatively small value (or even 0), the percentage of rQCs formed among all QCs, perhaps surprisingly, is large. This is because while a replica is packing the proposal, more signatures may have been received and an rQC can be formed. Note, however, a too small Δ_2 may lead to a low percentage of rQCs formed and increased latency. Also note that while a too small Δ_2 may trigger view change, the view change in Dashing1 is as efficient as that in HotStuff. Additionally, we require that after a view change, the new leader needs to form three consecutive rQCs; doing so allows all corresponding blocks for wQCs accumulated in the prior view to be committed. We also *optionally* limit the number of consecutive wQCs for the normal case to avoid unnecessary view changes. Namely, we can add an additional rule such that a leader needs to form three consecutive rQCs once every k (say, 50) wQCs have been extended on the branch.

Note that an overly large Δ_2 does not cause any (performance) issues, as the leader will propose a new block once $n - f$ votes are received. Namely, even if we set an overly large Δ_2 , Dashing1 would remain at least as efficient as HotStuff and Dashing1 remains optimistically responsive. Also, we comment that in settings where there exist natural network discrepancies, we set Δ_2 according to concrete network connection conditions.

The last challenge is to maintain certificates with two thresholds. If favoring maintaining linear authenticator complexity using threshold signatures, one should setup two threshold signature schemes—one for wQCs and the other for rQCs. In each round-trip communication, replicas should generate both a partial signature for wQC and a partial signature for rQC. The leader should maintain two sets storing threshold signatures for wQC and rQC, respectively. In a different approach, one can simply use conventional signatures and track all valid signatures in a single set. In our implementation, we adopt the second approach that uses conventional signatures, one also used in a series of HotStuff libraries [2, 24, 44, 52].

3.3 Notation for Dashing Protocols

Blocks. A block b is of the form $\langle req, pl, sl, view, height \rangle$. We use $b.x$ to represent the element x in block b . Fixing a block b , $b.pl$ is the hash digest of b 's parent block, $b.height$ is the number of blocks on the branch led by b , and $b.view$ is the view in which b is proposed. Note that different from the prior notation, sl is a new element in b . Formally, $b.sl$ denotes the hash digest of b 's stable block (the highest block with a regular certificate on the branch led by b). For simplicity, we also use $b.parent$ and $b.stable$ to represent the parent block and the stable block of b , respectively.

Messages. Messages transmitted among replicas are of the form $\langle TYPE, block, justify \rangle$. We use three message types—GENERIC, VIEW-CHANGE, and NEW-VIEW. GENERIC messages are used in normal operations. VIEW-CHANGE and NEW-VIEW messages are used during view change: VIEW-CHANGE messages are sent by replicas to the next leader, while NEW-VIEW messages are sent by the new leader to the replicas. The *justify* field stores certificates to validate the *block*. Fields may be set as \perp .

Functions and notation for QCs. A QC for message m is also called a QC for $m.block$. Fixing a QC qc for a block b , let $QC_{BLOCK}(qc)$ return the block b .

To hide implementation details of the QCs, we let $QC_{VOTE}(m)$ denote the output of a partial signature signing algorithm for m or a conventional signing algorithm and let $QC_{CREATE}(M)$ be a QC generated from signatures in M . $QC_{CREATE}(M)$ may be a wQC or an rQC.

Rank of QCs and blocks. Following the notion in [23], we now define the $rank()$ function for QCs and blocks. $rank()$ does not return a concrete number. Instead, it takes as input two blocks or QCs and outputs whether the rank of a block/QC is higher than the other one. The rank of two blocks/QCs is first compared by the view number, then by the height.

Local state at replicas. Each replica maintains the following state parameters, including the current view number $view$, the highest rQC QC_r , the highest wQC QC_w , the locked block lb , and the last voted block vb .

3.4 Dashing1

We present in Algorithm 2 and Algorithm 3 the normal case protocol and view change protocol of Dashing1, respectively. The utility functions are presented in Algorithm 1. We largely follow the description of HotStuff and highlight how Dashing1 supports wQCs in dotted boxes.

Algorithm 1: Utilities

```

1 procedure CREATEBLOCK( $b', v, req, qc$ )
2    $b.pl \leftarrow hash(b')$ ,  $b.parent \leftarrow b'$ ,  $b.height \leftarrow b'.height + 1$ 
3    $b.req \leftarrow req$ ,  $b.view = v$ 
4   if  $qc$  is a wQC then
5      $b.sl \leftarrow b'.sl$ ,  $b.stable \leftarrow b'.stable$ , return  $b$ 
6   if  $qc$  is an rQC then  $b.sl \leftarrow b.pl$ ,  $b.stable \leftarrow b'$  return  $b$ 
7   procedure STATEUPDATE( $QC_w, QC_r, lb, qc$ )
8      $b' \leftarrow QC_{BLOCK}(qc)$ ,  $b'' \leftarrow b'.parent$ ,  $b^* \leftarrow b''.parent$ ,
9      $v \leftarrow b'.view$ ,  $b_0 \leftarrow QC_{BLOCK}(QC_w)$ ,  $b_{high} \leftarrow QC_{BLOCK}(QC_r)$ 
10    if  $qc$  is an rQC then
11      if  $rank(b') > rank(b_{high})$  then  $QC_r \leftarrow qc$ 
12      if  $b'.stable = b''$  and  $rank(b'') > rank(lb)$  then  $lb \leftarrow b''$ 
13      if  $b'.stable = b''$  and  $b''.stable = b^*$  and
14         $b'.view = b^*.view = v$  then
15        deliver the transactions in  $b^*$  and ancestors of  $b^*$ 
16    if  $qc$  is a wQC and  $rank(b'.stable) \geq rank(b_0.stable)$  then
17       $QC_w \leftarrow qc$ 

```

Normal case protocol (Algorithm 2). We describe the chained version of the protocol. In each phase, the leader broadcasts a message and waits for signed responses from replicas. At lines 9-10, the leader first proposes a new block b and broadcasts a $\langle GENERIC, b, qc_{high} \rangle$ message, where qc_{high} is the last QC it receives (either a wQC or an rQC). The leader waits for the votes from the replicas. After collecting $f + 1$ matching votes, the leader starts a timer Δ_2 (ln 6) to determine whether the leader should stop waiting for more votes and propose a new block. Namely, the leader can propose a new block if either one of the two conditions is met: 1) Δ_2 expires; 2) it forms an rQC. After that, the leader combines the signatures in the votes into qc_{high} for the next phase.

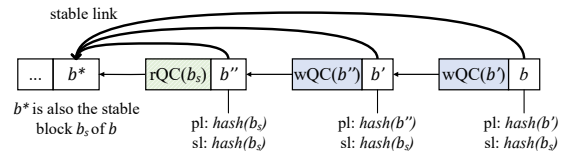


Figure 6. Illustration of the relationships of blocks in Algorithm 2.

Upon receiving a $\langle GENERIC, b, \pi \rangle$ message from the leader, each replica p_i first verifies whether b is well-formed (ln 13-16), i.e., b has a higher rank than its parent block b' and $b.height = b'.height + 1$. Let b'' denote the parent of b' , we distinguish two cases. For ease of understanding, we illustrate in Fig. 6 the relationships of b , b' , b'' , and b^* .

Algorithm 2: Normal case protocol of Dashing1 for

p_i

- 1 **initialization:** $cview \leftarrow 1$, vb , QC_w, QC_r, lb are initialized to \perp
- 2 Start a timer Δ_1 for the first request in the queue of pending transactions
- 3 **▷ GENERIC phase:**
- 4 **as a leader**
- 5 **wait for votes for b :**
 $M \leftarrow \{\sigma \mid \sigma \text{ is a signature for } \langle \text{GENERIC}, b, \perp \rangle\}$
- 6 **upon** $|M| = f + 1$ **then** set a start timer Δ_2
- 7 **upon** Δ_2 *timeout* or receiving $n - f$ matching messages **then**
- 8 $qc_{high} \leftarrow \text{QCCREATE}(M)$
- 9 $b \leftarrow \text{CREATEBLOCK}(b, cview, req, qc_{high})$
- 10 broadcast $m = \langle \text{GENERIC}, b, qc_{high} \rangle$
- 11 **as a replica**
- 12 **wait for** $m = \langle \text{GENERIC}, b, \pi \rangle$ from LEADER($cview$)
- 13 $b' \leftarrow b.parent$, $b'' \leftarrow b'.parent$, $b_s \leftarrow b.stable$
- 14 $m \leftarrow \langle \text{GENERIC}, b, \perp \rangle$
- 15 **if** $rank(b') \geq rank(b)$ **or** $b.height \neq b'.height + 1$
- 16 **discard the message**
- 17 **if** π is a wQC for b' **and** $b_s = b'.stable$ **and**
 $b_s.view = b'.view = b''.view = cview$ **and** $b' = vb$ **then**
 $vb \leftarrow b$, STATEUPDATE(QC_w, QC_r, lb, π)
- 18 **if** π is an rQC for b' **and** $b_s = b'$ **and** $rank(b') \geq rank(vb)$
- 19 $vb \leftarrow b$, STATEUPDATE(QC_w, QC_r, lb, π)
- 20 **if** $vb = b$ **then** send QCVOTE(m) to LEADER($cview$)
- 21 **▷ NEW-VIEW phase:** switch to this line if Δ_1 *timeout* occurs
- 22 **as a replica**
- 23 $cview \leftarrow cview + 1$
- 24 **send** $\langle \text{VIEW-CHANGE}, \perp, (QC_r, QC_w) \rangle$ to LEADER($cview$)

- If the π field is a wQC for b' (ln 17), p_i verifies if the stable block of b and b' are the same block such that b indeed extends b' . p_i also verifies if b , b' and b'' are all proposed in the same view and p_i has previously voted for b' . If so, p_i updates its local parameter QC_w to π . (Algorithm 1, ln 15).
- If π is an rQC for b' (ln 18-19), p_i verifies if b 's parent block b' has a higher rank than vb . If so, p_i updates its QC_r to π and generates a signature. If b'' has an rQC and b'' has a higher rank than the locked block of p_i , then p_i updates its lb to b'' . If p_i has received an rQC for both b'' and b^* (the parent block of b''), then p_i commits block b^* and delivers the transactions in b^* (Algorithm 1, ln 6-14).

In both cases, the replica updates its vb to b , and sends its vote (a signature for m) to the leader (ln 20).

View change protocol (Algorithm 3). Every replica starts timer Δ_1 for the first transaction in its queue. If the transaction is not processed before Δ_1 expires, the replica triggers view change. In particular, the replica sends a $\langle \text{VIEW-CHANGE}, \perp, (QC_r, QC_w) \rangle$ message to the new leader (Algorithm 2, ln 23-24). Upon receiving $n - f$ VIEW-CHANGE messages, the leader first obtains a block b_1 with an rQC that has the highest rank (ln 5). The leader then obtains a block

Algorithm 3: View change protocol of Dashing1 for

p_i

- 1 **▷ VIEW-CHANGE phase**
- 2 **as a new leader**
- 3 $//M$ is a set of $n - f$ VIEW-CHANGE messages collected by the new leader
- 4 $qc_{high} \leftarrow$ the rQC of highest rank contained in M
- 5 $b_1 \leftarrow \text{QCBLOCK}(qc_{high})$
- 6 **for** $m \in M$
- 7 **if** a wQC $qc_d \in m.justify$ **and** $\text{QCBLOCK}(qc_d) = d$ **and**
 $rank(d.stable) > rank(b_0.stable)$ **then** $vc \leftarrow qc_d$, $b_0 \leftarrow d$
- 8 **if** $rank(b_0.stable) \geq rank(b_1)$ **then**
 $b \leftarrow \text{CREATEBLOCK}(b_0, cview, req, vc)$,
broadcast $m = \langle \text{GENERIC}, b, vc \rangle$
- 9 **else then**
- 10 $b \leftarrow \text{CREATEBLOCK}(b_1, cview, req, qc_{high})$
- 11 broadcast $m = \langle \text{GENERIC}, b, qc_{high} \rangle$
- 12 $//switch$ to normal case protocol
- 13 **as a replica**
- 14 **wait for** $m = \langle \text{GENERIC}, b, \pi \rangle$ from LEADER($cview$)
- 15 $b' \leftarrow b.parent$, $b_s \leftarrow b.stable$, $m \leftarrow \langle \text{GENERIC}, b, \perp \rangle$
- 16 **if** $b'.view \geq cview$ **or** $rank(b') \geq rank(b)$ **or**
 $b.height \neq b'.height + 1$ **then**
- 17 **discard the message**
- 18 **if** π is a wQC for b' **and** $b_s = b'.stable$ **and** $rank(b_s) \geq$
 $rank(lb)$ **then** $vb \leftarrow b$, STATEUPDATE(QC_w, QC_r, lb, π)
- 19 **if** π is an rQC for b' **and** $b_s = b'$ **and** $rank(b_s) \geq rank(lb)$
- 20 **then** $vb \leftarrow b$, STATEUPDATE(QC_w, QC_r, lb, π)
- 21 **if** $vb = b$ **then** send QCVOTE(m) to LEADER($cview$)
- 22 $//switch$ to normal case protocol. Three consecutive rQCs are required for the first block proposed during the view change.
- 23 **▷ NEW-VIEW phase:** switch to NEW-VIEW phase if Δ_1 times out

b_0 with a wQC vc such that among all the blocks with weak QCs, b_0 has the highest stable block (first part of ln 6). Then the leader checks if the rank of the stable block of b_0 is no less than that of b_1 (second part of ln 6). If so, the leader creates a new block b extending b_0 and broadcasts b to all replicas. Otherwise, the leader extends b_1 and creates block b and broadcasts to the replicas (ln 5 and ln 9).

Upon receiving a $\langle \text{GENERIC}, b, \pi \rangle$ message from a new leader, each replica p_i verifies if the proposed block b extends a block of a prior view (ln 14-15). Then p_i votes for b if either of the following conditions is satisfied:

- b extends a block b' with a wQC (ln 16), the stable blocks of b and b' are the same block (denoted as b_s), and the rank of b_s is no less than that of the locked block of p_i ;
- b extends a block b' with an rQC (ln 17-18), and the rank of the stable block of b is no less than that of the locked block of p_i .

For the first block proposed in a new view, the leader needs to collect three consecutive rQCs after replicas switch to normal case protocol (ln 20). As discussed in Sec. 3.2, this is crucial for dealing with the liveness challenge caused

by the timer Δ_2 . Moreover, one may optionally enforce an additional rule such that the leader should commit at least one block after proposing "sufficient" blocks with wQCs (say, 50 wQCs).

State transfer. As in HotStuff, replicas in Dashing1 may need to perform state transfer with other replicas to obtain the QCs or transactions included in the QCs. For the state transfer of QCs, if a replica learns that a block b with height h is committed but it has not received any QCs between height h' of its locked block and h , the replica has to synchronize all the QCs for blocks between h' and h on the branch led by b . For the state transfer of the transactions for each QC, the replica needs to obtain the proposal from other replicas such that the hash of the proposal matches that in the QC.

3.5 Dashing2

We show in Dashing2 how to further enable a fast path using sQCs in Appendix B. We comment that such a task is more subtle than that in SBFT [26].

4 The Star Framework

We present Star, a new BFT framework that allows replicas to concurrently propose transactions and to deliver at least $n - f$ proposals in each epoch.

4.1 Overview of the Star Architecture

As in Narwhal and Tusk [1], the transmission and consensus processes in Star (as described in Fig. 7) are decoupled. The transmission process is fully parallelizable and works in asynchronous environments. It proceeds in epochs, where all replicas can propose transactions and output a queue of weak certificates numbered by epochs. The consensus process has only one BFT instance and does not carry bulk data. It takes as input weak certificates of the proposals and agrees on which proposals in each epoch should be delivered.

Compared to prior works, Star has the following distinctive features: 1) we use *pipelining* for the data transmission process to gain in efficiency; 2) Star uses wQCs for the data transmission process to further improve performance; 3) crucially, the transmission process and the consensus process are implicitly "correlated" with epoch numbers, and the consensus process only handles messages transmitted in the same epoch, which helps achieve effective censorship resilience and improve *blockchain quality*; 4) Star admits a more efficient and $O(1)$ time state transfer mechanism; and 5) Star has lower communication in both the gracious and uncivil scenarios than existing protocols of the same kind.

4.2 Star Details

The transmission process. The transmission process evolves in epochs. Each epoch consists of n parallel wCBC instances, as shown in Fig. 7 (a). Each replica maintains a queue Q of

pending transactions and outputs a growing set $W[e]$ containing weak certificates for each epoch e . In each wCBC instance, a designated replica broadcasts a proposal (a batch of transactions) from its queue of pending transactions. Upon completing $n - f$ wCBC instances, each replica starts the next epoch and continues to propose new transactions.

wCBC may be viewed as a weak version of consistent broadcast (CBC), i.e., CBC with weak certificates. A wCBC instance consists of three steps. First, a designated sender sends a proposal containing a set of transactions to all replicas. The sender waits for signed responses from $f + 1$ replicas to form a wQC and sends it to all replicas. Upon receiving a valid wQC, each replica delivers the corresponding proposal. Note it is possible that for a particular wCBC instance, a correct replica delivers m and another correct replica delivers $m' \neq m$. While multiple conflicting wQCs might be provided by a faulty sender, at most one wQC will be delivered.

So why wCBC? wCBC ensures that if a wQC is formed, at least one correct replica has received and stored the corresponding proposal. The use of wQCs is *sufficient* to ensure liveness, because any replica p_j , once obtaining wQC, can ask for the corresponding proposal from correct replicas; any correct replica that stores the proposal can simply send it to p_j , which can then validate the correctness of the proposal via the wQC. The above procedure is needed only when a correct replica stored a wQC but had no corresponding proposal. Even if the scenario occurs, it would not incur higher message or communication complexity.

Star develops the above idea and offers a pipelined version for high performance. Concretely, each replica can directly propose a new proposal in the third step of wCBC. We describe the code of the transmission process at ln 3-13 of Algorithm 4, where each replica p_i ($i \in [0..n - 1]$) runs the *initepoch*(e) function to start a new epoch e . Replica p_i chooses a set of transactions from Q as a proposal (say, b) using the *select* function. (The *select* function is vital to liveness and we will discuss its specification shortly.) It then broadcasts a message $\langle \text{PROPOSAL}, e, b, wqc \rangle$, where wqc is the wQC formed in epoch $e - 1$. (If we are working in the non-chaining mode, then wqc is simply \perp .) p_i waits for $f + 1$ votes for b to form a wQC. Then after receiving $n - f$ proposals for epoch e , p_i enters the next epoch $e + 1$. Upon receiving $\langle \text{PROPOSAL}, e, b_j, wqc_j \rangle$ from p_j , each replica first verifies wqc_j , sends a signed vote for b_j to p_j , adds b_j to *proposals*, and adds wqc_j to $W[e - 1]$.

The consensus process. The consensus process also proceeds in epochs, using only one BFT instance to agree on the wQCs. We can use any BFT protocol for the consensus process. When describing the consensus process (Algorithm 4, ln 15-20), we use the *a-broadcast* and *a-deliver* primitives.

Each replica p_i maintains le , a local parameter tracking the current consensus epoch number. p_i monitors its queue W (obtained from the transmission process) and checks whether

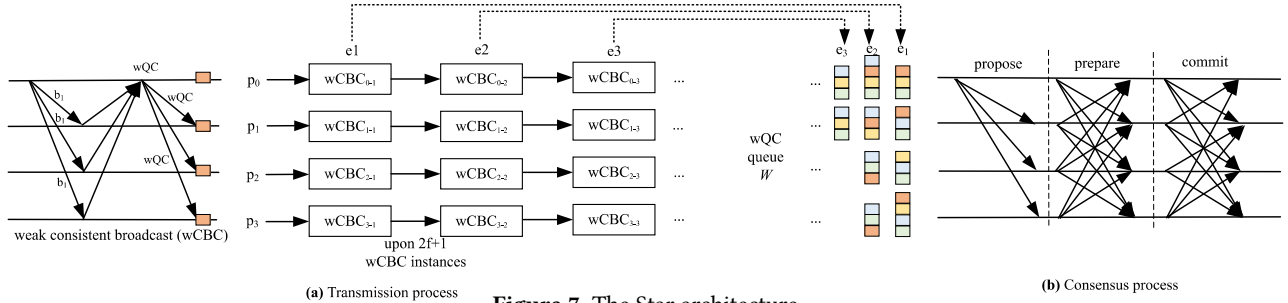


Figure 7. The Star architecture.

Algorithm 4: The code of Star for p_i

```

1 initialization: epoch number  $e$  and the epoch number of the
  current block  $le$  are initialized to 1. Queue  $Q$  of pending
  transactions, received proposals  $proposals$ , the latest weak
  certificate  $wqc$ , and queue  $W$  of weak certificates are initialized to  $\perp$ .
2 ▷ transmission process in the chaining (pipelined) mode
3 func  $initepoch(e)$ 
4    $b.tx \leftarrow select(Q)$ ,  $b.epoch \leftarrow e$  //select a proposal  $b$  from  $Q$ 
5   broadcast  $\langle PROPOSAL, e, b, wqc \rangle$ 
6   upon receiving a set  $M$  of  $f + 1$  signed votes for  $b$ 
7      $wqc \leftarrow QC CREATE(M)$  //create a weak certificate
8   wait until  $|proposals[e]| \geq n - f$  //enter the next epoch
9      $e \leftarrow e + 1$ ,  $initepoch(e)$ 
10 upon receiving  $\langle PROPOSAL, e, b_j, wqc_j \rangle$  from  $p_j$  for the first time
11   send signed vote for  $b_j$  to  $p_j$ 
12    $proposals[e] \leftarrow proposals[e] \cup b_j$ 
13    $W[e-1] \leftarrow W[e-1] \cup wqc_j$  //certificates in the output queue
14 ▷ consensus process
15 upon  $|W[le]| \geq n - f$ 
16    $a-broadcast(W[le])$  //run the underlying atomic broadcast
17 upon  $a-deliver(le, m)$ 
18    $O \leftarrow obtain(le, m)$ 
19    $deliver O$  //deliver the transactions in  $O$  in deterministic order
20    $le \leftarrow le + 1$ 
21 ▷ state transfer
22 func  $obtain(e, m)$ 
23    $O \leftarrow \perp$ 
24   for  $wQC qc \in m$ 
25     if  $QC PROPOSAL(qc) \in proposals[e]$ 
26        $O \leftarrow O \cup QC PROPOSAL(qc)$ 
27     else broadcast  $\langle FETCH, e, qc \rangle$ 
28       wait for a PROPOSAL containing  $QC PROPOSAL(qc)$ 
29        $O \leftarrow O \cup QC PROPOSAL(qc)$ 
30   clear  $W[e]$ , remove transactions in  $O$  from  $Q$ 
31 upon receiving message  $\langle FETCH, e, qc \rangle$  from replica  $p_j$ 
32   if  $QC PROPOSAL(qc) \in proposals[e]$  //fetch missing proposals
33     send  $\langle PROPOSAL, QC PROPOSAL(qc) \rangle$  to  $p_j$ 

```

$W[le]$ has at least $n - f$ weak certificates. If so, replicas run $a-broadcast(W[le])$. (If the underlying BFT is leader-based, then only the leader proposes $W[le]$). When the $a-deliver$ primitive terminates, each replica waits for the transactions corresponding to the a -delivered wQCs (from the transmission process) and delivers the transactions in deterministic

order. If some proposals are missing, the replica may simply fetch the proposals from other replicas (via the state transfer process at ln 21-33 of Algorithm 4). During state transfer, for each wQC qc in epoch e , a replica p_i broadcasts a $\langle FETCH, e, qc \rangle$ message to all replicas. Upon receiving such a message, a replica sends the corresponding proposal to p_i if it has the proposal.

Censorship resilience (liveness) and blockchain quality. Protocols allowing all replicas to propose different transactions should address transaction censorship which prevents a particular transaction proposed by a replica from never being delivered. First, the use of wQC ensures that if the underlying atomic broadcast completes, then the corresponding proposal has been obtained by correct replicas, or can be obtained via the fetch operation by correct replicas.

We should in addition ensure that adversary cannot censor certain transactions. So we have to be careful in specifying the $select$ function. HoneyBadgerBFT [42] invents a method where replicas randomly select transactions from their queue and use threshold encryption to achieve censorship resilience. EPIC [36] combines the conventional FIFO strategy used in [12] and the random selection strategy used in HoneyBadgerBFT to avoid threshold encryption. The asynchronous pattern in Star allows us to adopt the same approach as in EPIC, thereby achieving liveness under asynchrony.

Our work enforces a strong form of *blockchain quality*, ensuring that at least 1/2 of the total transactions contained in *any* committed block in an epoch are from correct replicas. Note that the concurrent work of Dumbo-NG does not satisfy this desirable feature: the quality of the multi-valued Byzantine agreement in Dumbo-NG does not lead to blockchain quality. Besides, Dumbo-NG requires unbounded memory.

Instantiating Star using PBFT. In Star, we use a variant of PBFT that is slightly different from PBFT. First, as the proposed transactions are already assigned with epoch number in the transmission process, we directly use the epoch number as the *sequence* number in the consensus process. We additionally require that the leader cannot skip any epoch number. Last, during a view change, the new leader is not allowed to propose a nil block for any epoch number. Namely, for any epoch e such that an agreement is not reached in a prior view, the new leader simply proposes $W[e]$.

Complexity analysis. Star has n parallel wCBC instances and one instance of the underlying BFT protocol (say, PBFT, HotStuff), so Star has $O(n^2)$ messages (whether using PBFT or HotStuff). The communication complexity is $O(Ln^2 + \lambda n^2)$ for the transmission process and $O(\lambda n^2)$ for the consensus process. As a replica can directly obtain a proposal based on epoch number and each QC, for state transfer of multiple QCs, the time complexity is $O(1)$.

In contrast, Narwhal changes the reliable broadcast used in DAG-Rider to consistent broadcast and the change leads to a complex and expensive state transfer mechanism. In particular, to obtain the transactions for round r , a replica has to perform state transfer to obtain the corresponding block. Then based on the block, the replica obtains the list of certificates (for round $r - 1$) and then needs to perform state transfer for the corresponding blocks. This process repeats until the replica obtains the entire causal history. Each of these steps has to be executed one after another since there is no guarantee that at least one correct replica holds the entire causal history. Therefore, if a replica needs to perform state transfer for k epochs in the transmission process, the time complexity is $O(k)$ while ours is $O(1)$. Moreover, Narwhal has $O(Ln^2 + \lambda n^3)$ communication, as each block consists of at least $2f + 1$ certificates of the prior round.

5 Implementation and Evaluation

We implement all our protocols introduced in this work and HotStuff in Golang using around 12,000 LOC, including 1,500 LOC for evaluation. We implement the chaining (pipelining) mode for the Dashing protocols and HotStuff. For all the protocols, we implement the checkpoint protocol for garbage collection, where replicas run the checkpoint protocol every 5000 blocks. We use gRPC as the underlying communication library. As in prior works [2, 24, 44, 52], we use digital signatures for quorum certificates. In particular, we use SM2 signature (ISO standard) which has similar performance as ECDSA. We also evaluate the performance of Narwhal [1].

We deploy the protocols in Amazon EC2 with up to 100 instances in both LAN and WAN. We use *m5.xlarge* instance which has four virtual CPUs and 16 GB memory. In the LAN setting, all the instances are located in the same region. In the WAN setting, the servers are eventually distributed over four different regions: us-west-1 (California, US), us-east-2 (Ohio, US), ap-southeast-1 (Singapore), and eu-west-1 (Ireland).

For each experiment, we use $3f + 1$ replicas and use f to denote the network size. We ask the clients to submit requests to the system in a non-closed loop, i.e., a client does not have to wait for the reply before sending the next request. We set the size for transactions and replies as 512 bytes. We focus on the throughput and latency of the protocols, where throughput is the number of transactions that can be delivered in each second and latency is the consensus time for each proposed block to be committed. We report

the average throughput and latency for each block to be committed during each experiment.

Performance (latency vs. throughput; throughput). We report the performance of Dashing1, Dashing2, Star, HotStuff, and Narwhal in both LAN and WAN settings.

In the LAN setting, we report latency vs. throughput for $f = 1$ and $f = 10$ in Fig. 8a and Fig. 8b and throughput as the number of clients increases in Fig. 8c and Fig. 8d. Dashing1 and Dashing2 consistently outperform HotStuff. For instance, the peak throughput of Dashing1 is 11.3% higher and 5.07% higher than HotStuff for $f = 1$ and $f = 10$, respectively. Meanwhile, Star significantly and consistently outperforms other protocols. For $f = 1$ and $f = 10$, the peak throughput of Star is 3.25x and 9.19x the throughput of HotStuff, respectively. Star also outperforms Narwhal consistently. Compared to that of Narwhal, the peak throughput of Star is 1.35% higher for $f = 1$ and 37.2% higher for $f = 10$.

In the WAN setting, we report the performance of the protocols in Fig. 8e-8l. Both Dashing protocols consistently outperform HotStuff, while Star significantly outpaces Narwhal. For instance, the peak throughput of Dashing1 is 107.36% higher and 49.8% higher than that of HotStuff for $f = 1$ and $f = 30$, respectively. Furthermore, when $f = 30$, Star achieves 35.9x the throughput of HotStuff, 23.9x that of Dashing1, and 2.38x that of Narwhal.

While Dashing1 and Dashing2 provide some interesting performance trade-offs, they offer similar throughput in most of the experiments. But Dashing2 has a fast path in the failure-free scenario, having lower latency in most cases.

Scalability. We report in Fig. 3 the peak throughput of Dashing1, Dashing2, Star, and HotStuff in the WAN environment as f grows. All the Dashing protocols outperform HotStuff consistently. The peak throughput of Dashing1 is 47%-107% higher than that of HotStuff. For the Dashing protocols and HotStuff, the throughput degrades as f grows, echoing other protocols in the HotStuff family. The throughput of Narwhal first increases as f grows and then decreases as f grows further, matching the evaluation result reported in Narwhal [1].

In comparison, the peak throughput of Star keeps growing as f increases (to 30). In particular, the performance of Star for $f = 30$ is 3.84x the throughput for $f = 1$. Meanwhile, the peak throughput of Star consistently outperforms other protocols. When $f = 30$, the peak throughput of Star is 243 ktx/sec, in contrast to 7 ktx/sec for HotStuff, 10 ktx/sec for Dashing1, and 102 ktx/sec for Narwhal. This is mainly because: 1) replicas only agree on a set of wQCs; 2) all n replicas propose transactions concurrently and the transmission process and consensus process are fully decoupled but implicitly correlated using epoch numbers; 3) the transmission process is highly efficient and can be pipelined. We comment that by asking the consensus processes to process the transactions transmitted with the same epoch number, Star can ensure $O(1)$ time delivery. This validates our motivation that wQCs can be used to achieve higher efficiency.

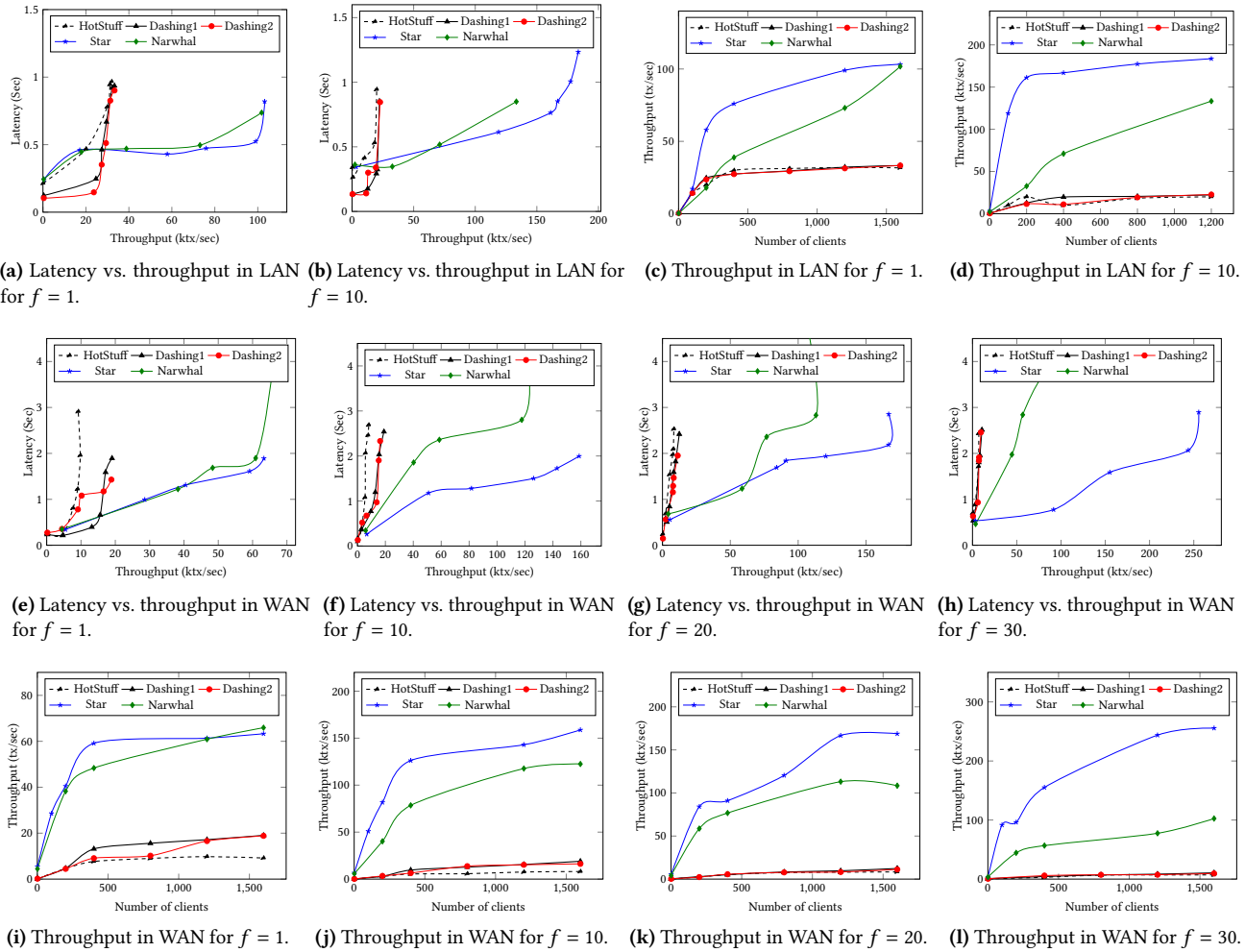


Figure 8. Performance of the protocols.

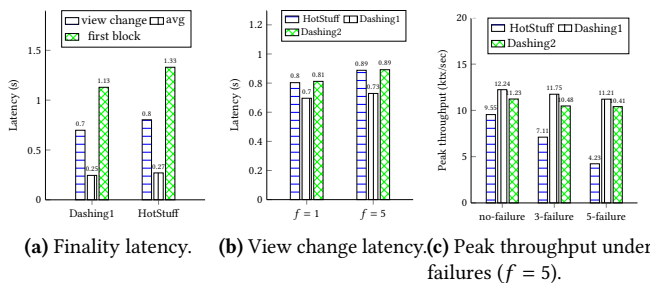


Figure 9. View change and performance under failures.

Performance under failures. We assess the performance under failures for Dashing1, Dashing2, and HotStuff. We use 1,200 clients in all the experiments.

We first assess the average latency of view changes due to the leader failures caused by halting the leader in the middle of each experiment. We report the view change latency for $f = 1$ and $f = 5$ in Fig. 9b. We find the view change latency for Dashing2 is higher than Dashing1 and HotStuff, mostly

because each NEW-VIEW message consists of $n - f$ messages and replicas need to verify them accordingly.

We also report the peak throughput for $f = 5$ with backup replica failures in Fig. 9c, where we evaluate the case of no failures, three failures, and five failures, respectively. For the case with three failures, we halt one server in each of the us-east-2, ap-southeast-1, and eu-west-1 regions. For the case with five failures, we halt two replicas in the eu-west-1 region, and one replica in each of the other three regions. In the case of failures, the performance of HotStuff degrades dramatically. In contrast, the Dashing protocols are highly robust against crash failures. Indeed, during failures, the leader in HotStuff always needs to collect $n - f$ votes from the remaining correct replicas; but the Dashing protocols can exploit wQCs and maintain consistent performance.

Dashing1 in operation. Dashing1 has been deployed in a major cross-border payment system with over 20 commercial banks involved. The system uses dedicated bank networking channels (called Direct Connect) for communication. While the network bandwidth varies for different sites, the average

| bandwidth | HotStuff | Dashing1 | improvement |
|-----------|----------|----------|-------------|
| 10 Mbps | 575 | 942 | 63.8% |
| 20 Mbps | 1,134 | 1,616 | 42.5% |
| 50 Mbps | 1,668 | 2,483 | 48.8% |
| 200 Mbps | 1,896 | 3,034 | 60.02% |

Table 2. Peak throughput of HotStuff and Dashing1 in operation.

bandwidth is 25.7 Mbps. (In contrast, the bandwidth in our Amazon EC2 experiments is significantly higher—around 10 Gbps.) Here we report the peak throughput with four sites (replicas) for Dashing1 and HotStuff with the following bandwidth settings: 10 Mbps, 20 Mbps, 50 Mbps, and 200 Mbps. Moreover, the machines used have 16-core CPU and 64GB memory, and the transaction size is 218 bytes. As shown in Table 2, while both Dashing1 and HotStuff achieve lower performance when compared to those conducted on EC2, Dashing1 consistently outperforms HotStuff, showing that weak certificates indeed lead to better performance.

6 Conclusion

We design and implement efficient BFT protocols using weak certificates, including a family of two Dashing protocols that offer improved efficiency and robustness compared to HotStuff, and a new (asynchronous) BFT framework Star allowing processing parallel transactions using a single BFT instance. Via a deployment in both the LAN and WAN environments, we show that our protocols outperform existing ones. In contrast to existing protocols, the throughput of Star keeps increasing as the number of replicas grows; in particular, in the WAN setting with 91 replicas across different continents, Star has a throughput of 243 ktx/sec, 35.9x the throughput of HotStuff and 2.38x the throughput of Narwhal.

References

- [1] Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus, author=Danezis, George and Kokoris-Kogias, Lefteris and Sonnino, Alberto and Spiegelman, Alexander, booktitle=Proceedings of the Seventeenth European Conference on Computer Systems, pages=34–50, year=2022.
- [2] HotStuff (Relab). <https://github.com/relab/hotstuff>, 2022.
- [3] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J.-P. Martin. Revisiting fast practical Byzantine fault tolerance. *arXiv preprint arXiv:1712.01367*, 2017.
- [4] I. Abraham, G. Gueta, D. Malkhi, and J.-P. Martin. Revisiting fast practical Byzantine fault tolerance: Thelma, Velma, and Zelma. *arXiv preprint arXiv:1801.10022*, 2018.
- [5] E. Androutaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *EuroSys*, 2018.
- [6] J.-P. Bahsoun, R. Guerraoui, and A. Shoker. Making BFT protocols really adaptive. In *IPDPS*, pages 904–913. IEEE, 2015.
- [7] A. Bessani, E. Alchieri, J. Sousa, A. Oliveira, and F. Pedone. From Byzantine replication to blockchain: Consensus is only the beginning. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 424–436. IEEE, 2020.
- [8] E. Blum, J. Katz, and J. Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In *Theory of Cryptography Conference*, pages 131–150. Springer, 2019.
- [9] E. Blum, C.-D. Liu-Zhang, and J. Loss. Always have a backup plan: fully secure synchronous MPC with asynchronous fallback. In *Annual International Cryptology Conference*, pages 707–731. Springer, 2020.
- [10] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the Gap-Diffie-Hellman-Group signature scheme. In *PKC*, pages 31–46, 2003.
- [11] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *Journal of cryptography*, 17(4):297–319, 2004.
- [12] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.
- [13] C. Cachin and M. Vukolić. Blockchain consensus protocols in the wild. In *DISC*, pages 1:1–1:16, 2017.
- [14] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *TOCS*, 20(4):398–461, 2002.
- [15] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, 2009.
- [16] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, pages 177–190. USENIX Association, 2006.
- [17] T. Distler, C. Cachin, and R. Kapitza. Resource-efficient Byzantine fault tolerance. *IEEE transactions on computers*, 65(9):2807–2819, 2015.
- [18] S. Duan, K. Levitt, H. Meling, S. Peisert, and H. Zhang. ByzID: Byzantine fault tolerance from intrusion detection. In *SRDS*, pages 253–264. IEEE, 2014.
- [19] S. Duan and H. Zhang. PACE: Fully parallelizable BFT from reproposable byzantine agreement. *IACR Cryptol. ePrint Arch.*, 2022.
- [20] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of ACM*, 32(2):288–323, 1988.
- [21] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. Technical report, Massachusetts Inst of Tech Cambridge lab for Computer Science, 1982.
- [22] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. Dumbo-NG: Fast asynchronous BFT consensus with throughput-oblivious latency. 2022.
- [23] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. *FC*, 2022.
- [24] N. Girdharan, H. Howard, I. Abraham, N. Crooks, and A. Tomescu. No-commit proofs: Defeating livelock in BFT. *Cryptology ePrint Archive*, 2021.
- [25] N. Girdharan, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. Bullshark: DAG bft protocols made practical. In *CCS*, 2022.
- [26] G. Golan-Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *DSN*, pages 568–580, 2019.
- [27] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. *ACM Transactions on Computer Systems*, 32(4):12:1–12:45, 2015.
- [28] M. Hirt, A. Kastrati, and C. Liu-Zhang. Multi-threshold asynchronous reliable broadcast and consensus. In *24th International Conference on Principles of Distributed Systems*, 2020.
- [29] M. M. Jalalzai, J. Niu, C. Feng, and F. Gai. Fast-Hotstuff: A fast and resilient Hotstuff protocol. *arXiv preprint arXiv:2010.11454*, 2021.
- [30] D. Kane, A. Fackler, A. Gagol, and D. Straszak. Highway: Efficient consensus with flexible finality. *arXiv preprint arXiv:2101.02159*, 2021.
- [31] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *EuroSys*, pages 295–308, 2012.
- [32] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman. All you need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of*

- Distributed Computing*, pages 165–175, 2021.
- [33] R. Kolta, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39, 2009.
- [34] K. Kursawe and V. Shoup. Optimistic asynchronous atomic broadcast. In *ICALP*, pages 204–215, 2005.
- [35] P. Kuznetsov and R. Rodrigues. Bftw3: Why? when? where? workshop on the theory and practice of byzantine fault tolerance. *ACM SIGACT News*, 40(4):82–86, 2010.
- [36] C. Liu, S. Duan, and H. Zhang. EPIC: Efficient asynchronous BFT with adaptive security. In *DSN*, 2020.
- [37] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic. XFT: Practical fault tolerance beyond crashes. In *OSDI*, pages 485–500, 2016.
- [38] Y. Lu, Z. Lu, and Q. Tang. Bolt-Dumbo transformer: Asynchronous consensus as fast as pipelined bft. In *CCS*, 2022.
- [39] D. Malkhi, K. Nayak, and L. Ren. Flexible Byzantine fault tolerance. In *CCS*, pages 1041–1053, 2019.
- [40] P. J. Marandi, M. Primi, and F. Pedone. High performance state-machine replication. In *DSN*, pages 454–465. IEEE, 2011.
- [41] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.
- [42] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of bft protocols. In *Proceedings of the SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.
- [43] A. Momose and L. Ren. Multi-threshold byzantine fault tolerance. In Y. Kim, J. Kim, G. Vigna, and E. Shi, editors, *CCS*, pages 1686–1699. ACM, 2021.
- [44] R. Neiheiser, M. Matos, and L. Rodrigues. Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation. In *SOSP*, pages 35–48, 2021.
- [45] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine-fault tolerance. *NSDI’09*, 2009.
- [46] J. Sousa, A. Bessani, and M. Vukolić. A Byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. In *DSN*, pages 51–58, 2018.
- [47] C. Stathakopoulou, T. David, and M. Vukolic. Mir-BFT: High-throughput BFT for blockchains. *arXiv preprint arXiv:1906.05552*, 2019.
- [48] C. Stathakopoulou, M. Pavlovic, and M. Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 17–33, 2022.
- [49] X. Sui, S. Duan, and H. Zhang. Marlin: Two-phase BFT with linearity. *DSN*, 2022.
- [50] M. Vukolić. Rethinking permissioned blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pages 3–7. ACM, 2017.
- [51] Z. Xiang, D. Malkhi, K. Nayak, and L. Ren. Strengthened fault tolerance in Byzantine fault tolerant replication. In *ICDCS*, pages 205–215. IEEE, 2021.
- [52] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *PODC*, 2019.

A Correctness of Dashing1

We first introduce some notation we use in this section. Let b', b denote two blocks such that $b.parent = b'$. According to Algorithm 2 and Algorithm 3, after receiving a GENERIC message $\langle \text{GENERIC}, b, qc \rangle$, a correct replica votes for b only if (1) $b.stable = b'$ and qc is an rQC for b' (ln 18-19 of Algorithm 2 and ln 17-18 of Algorithm 3); or (2) $b.stable = b'.stable$ and qc is a wQC for b' (ln 17 of Algorithm 2 and ln 16 of Algorithm 3). In both cases, we say that qc and b are *matching*.

Let b, b' and b'' denote three consecutive blocks. In Algorithm 1, we have that a replica p_i commits b only after receiving an rQC qc for b'' such that $b''.stable = b', b'.stable = b$, and $b.view = b'.view = b''.view = v$. In this case, we call qc a *commitQC* for b .

Lemma A.1. *If b and d are two conflicting blocks and $rank(b) = rank(d)$, then an rQC cannot be formed for both b and d .*

Proof. Let v denote $b.view$. As $rank(b) = rank(d)$, we have $d.view = v$. Suppose, towards a contradiction, an rQC is formed for both b and d . As a valid rQC consists of $2f + 1$ votes, a correct replica has voted for both b and d in view v . This causes a contradiction, because in the same view and for any height, a correct replica votes for at most one block. \square

Lemma A.2. *Suppose that there exists an rQC or a wQC qc for b ; if block d and d_c are on the branch led by b such that $d_c.parent = d$, then we have that*

- (1) $d.height < d_c.height$ and at least one correct replica has received a certificate qc_d for d , where qc_d and d_c are matching;
- (2) and if the view of the parent block of d is lower than $d.view$, then at least one correct replica has received an rQC qc_d for d and $d_c.stable = d$.

Proof. (1) We prove the claim (1) by induction for d . If $d = b.parent$, then d_c equals b . Since qc is an rQC or a wQC for b , at least one correct replica has voted for d_c . Then we have that $d.height < d_c.height$ and p_i has received a qc_d before voting for d_c , where qc_d and d_c are matching.

If $d \neq b.parent$, then there exists an rQC or a wQC for any block higher than d on the branch led by b . In this situation, there exists a block d_c on the branch led by b such that $d_c.parent = d$; an rQC or a wQC qc_c for d_c is received by at least one correct replica. Since qc_c consists of at least $f + 1$ votes, at least one correct replica p_i has voted for d_c in view $d_c.view$. Then we have that $d.height < d_c.height$ and p_i has received a qc_d before voting for d_c , where qc_d and d_c are matching. This completes the proof of claim (1).

(2) Based on claim (1), we know that at least one correct replica p_i has voted for d_c in view $d_c.view$. Let d' denote the parent block of b . Then $d'.view < d.view$. According to ln 16-18 of Algorithm 2, p_i votes for d_c only if p_i has received a rQC qc_d for d and $d_c.stable = d$. \square

Lemma A.3. *If there exists a wQC qc_d for block d , then d extends $d.stable$ and at least one correct replica has received a rQC for $d.stable$.*

Proof. Let d_0 denote $d.parent$. As there exists a wQC for d , at least one correct replica p_i has received a certificate qc and voted for d in view $d.view$, where qc and d are matching. We distinguish two cases:

- (1) qc is an rQC for d_0 and $d.stable = d_0$. Then we know that d extends $d.stable$, because d_0 is the parent block of d . Accordingly, at least one correct replica p_i has received a rQC qc for $d.stable$ before voting for d .

(2) qc is a wQC for d_0 and $d.stable = d_0.stable$. Let d_v denote the block of highest height on the branch led by d such that $d_v.stable \neq d.stable$. Let d'_v denote the block on the branch such that $d'_v.parent = d_v$. We have $d'_v.height > d_v$ and $d'_v.stable = d.stable$. Therefore, it follows from Lemma A.2 that at least one correct replica p_i has voted for d'_v . Thus, we have $d'_v.stable = d_v.stable$ or $d'_v.stable = d_v$ according to Algorithm 2 (ln 17-19). Since $d_v.stable \neq d.stable$, we have that $d'_v.stable \neq d_0.stable$. Then we know that $d'_v.stable = d.stable = d_v$ and d extends $d.stable$. Meanwhile, p_i has received an rQC for d_v before voting for d'_v .

In both cases, d extends $d.stable$ and a correct replica has received an rQC for $d.stable$. \square

Lemma A.4. *If there exists at least one rQC formed in view v , then there exists only one rQC qc of lowest rank in view v , and we have that*

(1) *The view of $b.parent$ is lower than v , where b equals $QCLOCK(qc)$;*

(2) *If there exists an rQC for b_1 and $b_1.parent.view < v$, then b_1 equals b .*

Proof. If an rQC is formed in view v , then there exists only one rQC qc of lowest rank in view v (according to Lemma A.1).

(1) Let b denote $QCLOCK(qc)$ and b_v denote the block of lowest height such that $b_v.view = v$ on the branch led by b . Therefore, $b_v.height \leq b.height$ and the view of $b_v.parent$ is lower than v . According to Lemma A.2, there must exist a rQC for b_v . Since qc is the lowest rQC formed in view v , we have that $b_v = b$ and the view of $b.parent$ is lower than v .

(2) If there exists an rQC for b_1 , then at least one correct replica has voted for b_1 and b in view v . Note that in view v , a correct replica only votes for one block that extends a block proposed in a lower view according to Algorithm 3. Therefore, it must hold that $b_1 = b$. \square

Lemma A.5. *If rQC qc for b is the rQC of lowest height formed in view v and there exists an rQC for block d such that $d.view = v$, then d equals b or d is an extension of b .*

Proof. Let d_0 denote the block of lowest height on the branch led by d such that $d_0.view = v$. Then the view of the parent block of d_0 is lower than v . According to Lemma A.2, at least one correct replica has received an rQC for d_0 . By Lemma A.4, it holds that d_0 equals b . As d_0 is a block on the branch led by d , d equals b or d is an extension of b . \square

Lemma A.6. *Suppose qc_1 and qc_2 are two rQCs, and each is received by at least one correct replica. Let b_1 and b_2 be $QCLOCK(qc_1)$ and $QCLOCK(qc_2)$, respectively. If b_1 is conflicting with b_2 , then $b_1.view \neq b_2.view$.*

Proof. Assume towards contradiction that $b_1.view = b_2.view = v$. According to Lemma A.5, we know that there exists a block b which is the block of lowest height for which an rQC was formed in view v , b_1 and b_2 are blocks and either b_1 or b_2

is equals b or is an extensions of b . Then $b_1.height \geq b.height$ and $b_2.height \geq b.height$. We consider three cases:

(1) If $b_1.height = b.height$ or $b_2.height = b.height$, then b_1 equals b or b_2 equals b . Therefore, b_1 and b_2 are the same block or they are on the same branch.

(2) If $b.height < b_1.height$, $b.height < b_2.height$, and $b_1.height = b_2.height$, then according to Lemma A.1, b_1 and b_2 must be the same block.

(3) If $b.height < b_1.height$, $b.height < b_2.height$, and $b_1.height \neq b_2.height$, then b_1 and b_2 are extensions of b . W.l.o.g., we assume that $b_1.height < b_2.height$. Let b'_2 denote a block on the branch led by b_2 such that $b'_2.height = b_1.height$. Then b'_2 is an extension of b . If b'_2 is conflicting with b_1 , then according to Lemma A.1, we have that no rQC for b'_2 can be formed in view v and at most f correct replicas voted for b'_2 . Thus, an rQC for any extensions of b'_2 cannot be formed by Algorithm 2. Therefore, we have that b'_2 must be equal to b_1 .

In all cases, b_1 and b_2 must be blocks on the same branch, contradicting the condition that they are conflicting blocks. Therefore, we have that $b_1.view \neq b_2.view$. \square

Lemma A.7. *If there exists a commitQC qc for b and an rQC qc_d for d , each is received by at least correct replica, and $rank(b) < rank(d)$, then d must be an extension of b .*

Proof. Let v be $b.view$, v_d be $d.view$, b'' be $QCLOCK(qc)$, and b' be $b''.parent$. As qc is a commitQC for b , we have that $b'.stable = b'.parent = b$, $b''.stable = b'$, and $b.view = b'.view = b''.view = v$. According to Lemma A.2, there exist rQCs for b , b' , and b'' such that all these rQCs are received by at least one correct replica. Note that an rQC for b' is also a lockedQC for b . Let S denote the set of correct replicas that have voted for b'' . Since qc consists of $2f + 1$ votes, we know that $|S| \geq f + 1$.

Since $rank(d) > rank(b)$, $v_d \geq v$. Then we prove the lemma by induction over the view v_d , starting from view v . **Base case:** Suppose $v_d = v$. According to Lemma A.6, d must be an extension of b .

Inductive case: Assume this property holds for view v_d from v to $v + k - 1$ for some $k \geq 1$. We now prove that it holds for $v_d = v + k$. Let b_0 denote the block of lowest height for which an rQC qc_0 was formed in view v_d and b'_0 denote $b_0.parent$. Let m denote the GENERIC message for b_0 . According to Lemma A.4, $b'_0.view < v_d$ and b_0 is proposed during view change. Since qc_0 consists of $2f + 1$ votes, at least one replica $p_i \in S$ has voted for b_0 in view v_d . Let b_{lock} denote the locked block lb of p_i when voting for b_0 . Note that p_i updates its lb only after receiving a lockedQC for a block of a higher rank than its locked block. Then we know that $rank(b_{lock}) \geq rank(b)$. Note that $b_{lock}.view < v_d$. According to Lemma A.6 and the inductive hypothesis, b_{lock} must be either equal to b or an extension of b . Then p_i votes for b_0 only if one of the following conditions is satisfied:

- 1) $b_0.stable = b'_0.stable$, $m.justfy$ is a wQC for b'_0 , $b'_0.view < v_d$ and $rank(b'_0.stable) \geq rank(b_{lock})$ (ln 16 in Algorithm 3).
- 2) $b_0.stable = b'_0$, $m.justfy$ is an rQC for b'_0 , $b'_0.view < v_d$, and $rank(b'_0) \geq rank(b_{lock})$ (ln 17 in Algorithm 3).

If condition 1) is satisfied, then according to Lemma A.3, b'_0 is an extension of $b'_0.stable$ and at least one correct replica has received an rQC for $b'_0.stable$. Note that $rank(b'_0.stable) \geq rank(b_{lock})$. According to Lemma A.1 and the inductive hypothesis, $b'_0.stable$ is equal to b or an extension of b . Hence, b_0 must be an extension of b .

If condition 2) is satisfied, then $rank(b'_0) \geq rank(b_{lock}) \geq rank(b)$ and $m.justfy$ is an rQC for b'_0 . According to Lemma A.1 and the inductive hypothesis, b'_0 is either equal to b or an extension of b .

Either way, b_0 must be an extension of b . Note that an rQC for d is formed in view v_d . According to Lemma A.5, we know that d is equal to b_0 or an extension of b_0 . Therefore, d must be an extension of b and the property holds in view $v+k$. This completes the proof of the lemma. \square

Theorem A.8. (safety) *If b and d are conflicting blocks, then they cannot be committed each by at least one correct replica.*

Proof. Suppose that a *commitQC* is formed for both b and d . According to Lemma A.2, there must exist rQCs for both b and d , each received by at least one correct replica. If $b.view = d.view$, then according to Lemma A.6, rQCs for both b and d cannot be formed. If $b.view \neq d.view$, w.l.o.g., we assume that $rank(b) < rank(d)$. According to Lemma A.7, an rQC for d cannot be formed in view $d.view$. Hence, no *commitQC* for d can be formed in view $d.view$. In both cases, *commitQC* for both b and d cannot be formed. \square

Theorem A.9. (liveness) *After GST, there exists a bounded time period T_f such that if the leader of view v is correct and all correct replicas remain in view v during T_f , then a decision is reached.*

Proof. Suppose after GST, in a new view v , the leader p_i is correct. Then p_i can collect a set M of $2f + 1$ VIEW-CHANGE messages from correct replicas and broadcast a new block b in a message $m = \langle \text{GENERIC}, b, qc \rangle$.

Let b' denote $b.parent$. Let b_{high} denote the block of highest rank locked by at least one correct replica. Note that a correct replica locks b_{high} only after receiving a *lockedQC* for it. Let b_1 denote $QCBlock(qc)$. Then we know that $b_1.parent = b_1.stable = b_{high}$ and a set S of at least $f + 1$ correct replicas have voted for b_1 . Therefore, at least one message in M is sent by a replica $p_j \in S$. According to Algorithm 2 and Algorithm 3, a correct replica votes for block b_1 only after receiving an rQC for b_{high} and QC_r of the replica is the rQC of highest rank received by the replica. Thus, the rank of the rQC qc_j sent in VIEW-CHANGE message by p_j is no less than that of b_{high} . From Algorithm 3, there are two cases for b : (1) $b.stable = b'$, qc is an rQC for b' and

$rank(qc) \geq rank(qc_j)$; (2) $b.stable = b'.stable$, qc is a wQC for b' and $rank(b'.stable) \geq rank(qc_j)$. In case (1), b will be voted for by all the correct replicas as conditions on ln 17 of Algorithm 3 are satisfied. In case (2), b will be voted for by all the correct replicas as conditions on ln 16 of Algorithm 3 are satisfied.

If all correct replicas are synchronized in their view, p_i is able to form a *QC* for b and generate new blocks. All correct replicas will vote for the new blocks proposed by p_i . Therefore a *commitQC* for b can be formed by p_i , leading to a new decision. Hence, after GST, the duration T_f for these phases to complete is of bounded length. This completes the proof of the theorem. \square

B Dashing2

B.1 Overview of Dashing2

Intuitively, supporting a $3f + 1$ threshold may allow replicas to deliver the transactions in a single phase: if the leader collects an sQC for a block and broadcasts to the replicas, replicas can directly commit the block.

While prior works have demonstrated how to design secure BFT protocols using strong quorums [3, 4, 26], integrating sQCs in Dashing1, however, has its unique challenges due to usage of wQCs. Indeed, as a block supported by an sQC may be extended from a block with only a weak certificate, replicas cannot directly commit the block upon receiving an sQC. As depicted in Fig. 10, two conflicting blocks b and b' are proposed in the same view 1 with the same height. Moreover, an rQC is formed for b and a wQC is formed for b' . Besides, a wQC for block b'_1 that extends b'_0 is formed. Suppose now a view change occurs, the new leader in view 2 extends b'_1 and proposes b'_2 . In Dashing1, replicas can vote for b'_2 , so an sQC can be formed. Then we consider a scenario where another view change occurs and replicas enter view 3. As there is no guarantee on how many correct replicas have received the sQC for b'_2 , the new leader in view 3 may choose to extend b_0 . And b_0 can be later committed in view 3, in which case safety is violated as b'_2 is committed in view 2. As a view change may occur at any moment, replicas cannot directly commit a block when an sQC is received.

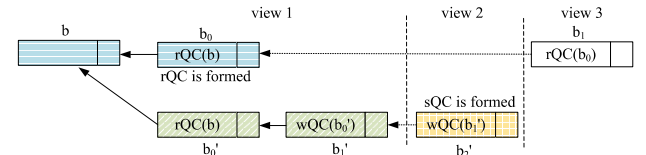


Figure 10. Challenge of integrating strong certificates in Dashing2.

In Dashing2, we treat an sQC for the first block proposed after view change as an rQC and the block cannot be committed immediately. Furthermore, during the view change, the new leader needs to send the VIEW-CHANGE messages from the replicas to all replicas, serving as a *proof* for the

block it proposes. In fact, the view change protocol now becomes similar to that in Fast-HotStuff [29] and Jolteon [23]. Accordingly, Dashing2 has $O(n^2)$ authenticator complexity and $O(n)$ message complexity. In addition, Dashing2 is a two-phase protocol with a one-phase fast path.

We make several major changes on top of Dashing1. First, Dashing2 follows the two-phase commit rule that if a replica receives an rQC for both a block b and b' (the parent block of b), block b' can be committed. Second, if a replica p_i receives an sQC for block b from the leader in normal case operation, p_i directly commits b and delivers the transactions unless b extends a block with a wQC. Third, during view change, the NEW-VIEW message from the new leader includes a set of at least $n - f$ VIEW-CHANGE messages. Upon receiving the NEW-VIEW message with a proposal, a correct replica verifies the proposal by performing a computation as the one used by the new leader to create the proposal. Replicas resume normal operations only after the NEW-VIEW message is verified. Finally, for the first block b proposed after each view change, the leader forms an rQC rather than wQC or sQC for b to start the normal case operations.

Note that as in BFT protocols using strong quorums [3, 4, 26], Dashing2 does not achieve optimistic responsiveness (which is unavoidable due to the one-phase fast path).

B.2 Dashing2 Details

Compared with Dashing1, an sQC is used as a certificate for a fast path in Dashing2. We present in Algorithm 6 and Algorithm 7 the normal case operation and view change protocol of Dashing2, respectively. The utility functions are presented in Algorithm 5. Dashing2 follows the notation of Dashing1. rQCs and sQCs are collectively called *qualified* QCs in this section.

Normal case protocol (Algorithm 6). Similar to Dashing1, in each phase, the leader broadcasts a block b in message $\langle \text{GENERIC}, b, qc_{high} \rangle$ to all replicas and waits for signed responses from the replicas. qc_{high} is the last QC the leader receives (either a wQC, an rQC, or an sQC). After collecting $f + 1$ matching votes, the leader starts a timer Δ_2 (ln 6). The timer is used to determine if the leader can form an rQC or an sQC in time. After Δ_2 expires, the leader combines the signatures in the votes into qc_{high} for the next phase.

Upon receiving a $\langle \text{GENERIC}, b, \pi \rangle$ message from the leader, each replica p_i first verifies whether b is well-formed and proposed during normal operation (ln 16-17), i.e., b has a higher rank than its parent block b' , $b.height = b'.height + 1$, and b' and b are proposed in the same view. Let b'' denote the parent of b' . We distinguish two cases:

- If the π field is a wQC for b' (ln 19-22), p_i verifies if the stable block of b and b' are the same block such that b indeed extends b' . p_i also verifies if b , b' , b'' , and $b.stable$ are all proposed in the same view and p_i has previously

voted for b' . If so, p_i updates its local parameter QC_w to π and creates a signature for b (Algorithm 5, ln 13).

- If π is an rQC or an sQC for b' (ln 23-25), p_i verifies if the stable block of b is b' , b' does not have a lower rank than vb , and b' does not have a lower rank than the QC_r of p_i . If so, p_i updates its local parameter QC_r to π and generates a signature (Algorithm 5, ln 10 and ln 15). If π is an rQC, b'' has a qualified QC, and b'' and b are proposed in the same view, then p_i commits block b'' and delivers transactions in b'' (Algorithm 5, ln 11-12). If π is an sQC, b'' has a qualified QC, and b'' and b are proposed in the same view, then p_i commits block b' and delivers the transactions in b' (Algorithm 5, ln 14-15).

In both cases, the replica updates its vb to b , and sends its signature to the leader.

View change protocol (Algorithm 7). Every replica starts timer Δ_1 for the first transaction in its queue. If the transaction is not processed before Δ_1 expires, the replica triggers view change. In particular, the replica sends a $\langle \text{VIEW-CHANGE}, vb, (QC_r, QC_w) \rangle$ message to the leader (Algorithm 6, ln 30). Upon receiving $n - f$ VIEW-CHANGE messages (denoted as M), the leader chooses a block to extend based on the output of $\text{SAFEBLOCK}(M)$ in Algorithm 5.

We now describe the procedure in more detail. Below, all number of lines is referred to as that in Algorithm 5. First, the leader obtains a block b_1 with a QC that has the highest rank (ln 17-18). The leader then obtains a block b_0 with a wQC vc such that b_0 , $b_0.parent$ and $b_0.stable$ are proposed in the same view, and among all the blocks with weak QCs, b_0 has the highest stable block (ln 19-24). The leader also obtains block b_2 such that b_2 is contained in more than $f + 1$ VIEW-CHANGE messages in M . If no such block exists, b_2 is set to \perp (ln 18 and ln 25-26). Then the leader checks if the rank of the stable block of b_2 is no less than that of b_1 (ln 27). If so, the leader selects b_0 to extend. Otherwise, the leader checks if the rank of the stable block of b_0 is no less than that of b_1 (ln 28). If so, the leader will extend b_0 . If neither is satisfied, the leader chooses b_1 to extend (ln 29).

Then the leader extends the selected block with a block b and broadcasts b to the replicas (ln 4-5 of Algorithm 7).

Upon receiving a $\langle \text{NEW-VIEW}, b, M \rangle$ message from a new leader, each replica p_i verifies b basing on the output of $\text{SAFEBLOCK}(M)$ (ln 14-18). If b is a block extending the output block of $\text{SAFEBLOCK}(M)$, then p_i votes for b (ln 16 and ln 18).

B.3 Correctness of Dashing2

We first introduce some notation we use for the proof. Let b' and b denote two blocks such that $b.parent = b'$ and $b.view = b'.view$. According to Algorithm 6, after receiving a GENERIC message $\langle \text{GENERIC}, b, qc \rangle$, a correct replica votes for b only if (1) $b.stable = b'$ and qc is an rQC or an sQC for b' (ln 23-25); or (2) $b.stable = b'.stable$ and qc is a wQC for b' (ln 19-22). In both cases, we say that qc and b are *matching*.

Algorithm 5: Utilities for Dashing2

```

1 procedure CREATEBLOCK( $b', v, cmd, qc$ )
2    $b.pl \leftarrow hash(b'), b.parent \leftarrow b', b.req \leftarrow req,$ 
3    $b.height \leftarrow b'.height + 1, b.view \leftarrow v$ 
4   if  $qc$  is a wQC or  $\perp$  then  $b.sl \leftarrow b'.sl, b.stable \leftarrow b'.stable,$ 
   return  $b$ 
5   if  $qc$  is an rQC or an sQC then  $b.sl \leftarrow hash(b')$  return  $b$ 
6 procedure STATEUPDATE( $QC_w, QC_r, qc$ )
7    $b' \leftarrow QCBLOCK(qc), b'' \leftarrow b'.parent,$ 
8    $b_0 \leftarrow QCBLOCK(QC_w), b_{high} \leftarrow QCBLOCK(QC_r)$ 
9   if  $qc$  is an rQC
10     $QC_r \leftarrow qc$ 
11    if  $b'.stable = b''$  and  $b''.view = b'.view$  then
12      deliver the transactions in  $b''$ 
13    if  $qc$  is a wQC then  $QC_w \leftarrow qc$ 
14    if  $qc$  is an sQC and  $b'.stable = b''$  and  $b'.view = b''.view$ 
15      then  $QC_r \leftarrow qc,$  deliver the transactions in  $b'$ 
16 procedure SAFEBLOCK( $M$ )
17    $qc_{high} \leftarrow$  the qualified QC of highest rank contained in  $M$ 
18    $b_1 \leftarrow QCBLOCK(qc_{high}), b_2 \leftarrow \perp, b \leftarrow$ 
   CREATEBLOCK( $b_1, cview, req, qc_{high}$ )
19   for a wQC  $qc \in M.justify$ 
20      $d \leftarrow QCBLOCK(qc), d' \leftarrow d.parent, d_s \leftarrow d.stable$ 
21     if  $d_s.view = d'.view = d.view$ 
22       if  $rank(d_s) > rank(b_0.stable)$  then  $vc \leftarrow qc, b_0 \leftarrow d$ 
23       if  $rank(d_s) = rank(b_0.stable)$  and  $rank(d) >$ 
24          $rank(b_0)$  then  $vc \leftarrow qc, b_0 \leftarrow d$ 
25   for  $d \in M.block$ 
26     if  $num(d, M.block) \geq f + 1$  then  $b_2 \leftarrow d$ 
27   if  $rank(b_2.stable) \geq rank(b_1)$  then return ( $b_2, \perp$ )
28   else if  $rank(b_0.stable) \geq rank(b_1)$  then return ( $b_0, vc$ )
29   return ( $b_1, qc_{high}$ )

```

Let b' and b denote two consecutive blocks. In Algorithm 5, a replica p_i commits b only after receiving a certificate qc and one of the following conditions is satisfied:

- (1) qc is an rQC for b' such that $b'.stable = b'.parent = b$ and $b.view = b'.view$ (ln 9-12);
- (2) qc is an sQC for $b, b.stable = b.parent$ and $b.parent.view = b.view$ (ln 14-15).

In both cases, qc is a *commitQC* for b .

Lemma B.1. *Suppose a block b has been voted for by a correct replica, then*

- (1) *any block d on the branch led by b has been voted for by at least one correct replica and $d.parent.height + 1 = d.height$;*
- (2) *if d and d_c are two blocks on the branch led by b such that $d_c.parent = d$ and $d_c.view = d.view = v$, then we have that (i) at least one correct replica has received a certificate (wQC, rQC, or sQC) qc_d for d , where qc_d and d_c are matching; (ii) if the view of the parent block of d is lower than v , then at least one correct replica has received a qualified QC for d and $d_c.stable = d$.*

Proof. Let d denote a block on the branch led by b .

(1) We prove claim (1) by induction for d . If $d = b$, then d has been voted for by at least one correct replica.

Algorithm 6: Normal case protocol for Dashing2

```

1 initialization:  $cview \leftarrow 1, vb, QC_w, QC_r \leftarrow \perp$ 
2 Start a timer  $\Delta_1$  for the first request in the queue of pending
   transactions
3 > GENERIC phase:
4 as a leader
5   wait for votes for  $b: M \leftarrow \{\sigma | \sigma \text{ is a signature for}$ 
    $\langle \text{GENERIC}, b, \perp \rangle\}$ 
6   upon  $|M| = f + 1$  then set a start timer  $\Delta_2$ 
7   upon  $\Delta_2$  timeout then  $qc_{high} \leftarrow QCCREATE(M)$ 
8      $b \leftarrow CREATEBLOCK(b, cview, cmd, qc_{high})$ 
9     broadcast  $m = \langle \text{GENERIC}, b, qc_{high} \rangle$ 
10    if  $qc_{high}$  is a wQC then  $QC_w \leftarrow qc_{high}$ 
11    if  $qc_{high}$  is an rQC or an sQC then  $QC_r \leftarrow qc_{high}$ 
12 as a replica
13   wait for  $m = \langle \text{GENERIC}, b, \pi \rangle$  from LEADER( $cview$ )
14    $b' \leftarrow b.parent, b'' \leftarrow b'.parent, b_s \leftarrow b.stable,$ 
15    $b_{gen} \leftarrow QCBLOCK(QC_r), m \leftarrow \langle \text{GENERIC}, b, \perp \rangle$ 
16   if  $rank(b') \geq rank(b)$  or  $b.height \neq b'.height + 1$  or
17      $b'.view \neq cview$  or  $b.view \neq cview$  then
18     discard the message
19   if  $\pi$  is a wQC for  $b'$  and  $b.sl = b'.sl$  and  $rank(b_s) \geq$ 
20      $rank(b_{gen})$  and  $b_s.view = b''.view = b'.view = cview$ 
21     and  $b' = vb$  then
22      $vb \leftarrow b, STATEUPDATE(QC_w, QC_r, \pi)$ 
23   if  $\pi$  is an rQC or an sQC for  $b'$  and  $b.stable = b'$ 
24     and  $rank(b') \geq rank(vb)$  and  $rank(b') \geq rank(b_{gen})$ 
25      $vb \leftarrow b, STATEUPDATE(QC_w, QC_r, \pi)$ 
26   if  $vb = b$  then send QCVote( $m$ ) to LEADER( $cview$ )
27 > NEW-VIEW phase: switch to this line if  $\Delta_1$  timeout occurs
28 as a replica
29    $cview \leftarrow cview + 1$ 
30   send  $\langle \text{VIEW-CHANGE}, vb, (QC_r, QC_w) \rangle$  to LEADER( $cview$ )

```

If $d \neq b$ and any block higher than d on the branch led by b has been voted for by at least one correct replica, then we need to prove that d is voted for by at least one correct replica. In this situation, there exists a block d_c on the branch led by b such that $d_c.parent = d$ and d_c has been voted for by at least one correct replica p_i . According to Algorithm 2 and Algorithm 3, $rank(d) < rank(d_c)$ and $d_c.height = d.height + 1$. Therefore, $d.view \leq d_c.view$.

We now distinguish two cases: $d.view = d_c.view$ and $d.view < d_c.view$.

If $d.view = d_c.view$, then p_i has received a qc_d for d , where qc_d and d_c are matching according to Algorithm 6. As qc_d consists of at least $f + 1$ votes, at least one correct replica has voted for d and $d.parent.height + 1 = d.height$.

If $d.view < d_c.view$, then from Algorithm 7 we know that d_c is proposed in a NEW-VIEW message m in view $d_c.view$ and $m.justify$ contains a set M of $2f + 1$ VIEW-CHANGE messages for view $d_c.view$. Then p_i votes for d_c if (i) a wQC, an rQC or an sQC for d is provided by a replica in M , or (ii) for $f + 1$ messages in M , the *block* fields are all set to d . In either case,

Algorithm 7: View change protocol for Dashing2

```

1 ▷ VIEW-CHANGE phase
2 as a new leader
3 //M is a set of  $n - f$  VIEW-CHANGE messages collected by the new
  leader
4  $(b', qc) \leftarrow \text{SAFEBLOCK}(M), b \leftarrow$ 
   $\text{CREATEBLOCK}(b', \text{cview}, \text{cmd}, qc)$ 
5 broadcast  $m = \langle \text{NEW-VIEW}, b, M \rangle$ 
6 //switch to normal case protocol
7 as a replica
8 wait for  $m = \langle \text{NEW-VIEW}, b, \pi \rangle$  from  $\text{LEADER}(\text{cview})$ 
9  $b' \leftarrow b.\text{parent}, b_s \leftarrow b.\text{stable}, b_{\text{gen}} \leftarrow \text{QCBLOCK}(QC_r),$ 
10  $m \leftarrow \langle \text{GENERIC}, b, \perp \rangle$ 
11 if  $b'.\text{view} \geq \text{cview}$  or  $\text{rank}(b') \geq \text{rank}(b)$  or  $b.\text{height} \neq$ 
12  $b'.\text{height} + 1$  then discard the message
13 if  $M \in \pi$ 
14  $(b_p, qc) \leftarrow \text{SAFEBLOCK}(M), m \leftarrow \langle \text{GENERIC}, b, \perp \rangle$ 
15 if  $b_p = b'$  and  $qc$  is a wQC or  $\perp$  and  $b.\text{stable} = b'.\text{stable}$ 
16 then send  $\text{QCVote}(m)$  to  $\text{LEADER}(\text{cview})$ 
17 if  $b_p = b'$  and  $qc$  is an rQC or sQC and  $b.\text{stable} = b'$ 
18 then send  $\text{QCVote}(m)$  to  $\text{LEADER}(\text{cview})$ 
19 //switch to normal case protocol
20 ▷ NEW-VIEW phase: switch to NEW-VIEW phase if  $\Delta_1$  timeout
  occurs

```

d has been voted for by at least one correct replica. This completes the proof of claim (1).

(2) Based on claim (1), at least one correct replica p_i has voted for d_c . (i) If $d_c.\text{view} = d.\text{view} = v$, then d_c is proposed during normal case operation. According to ln 19 and ln 23 of Algorithm 6, p_i has received a certificate (wQC, rQC, or sQC) qc_d for d before voting for d_c , where d and d_c are matching. (ii) Meanwhile, according to ln 16-25 of Algorithm 6, if $d.\text{parent.view} < v$, then p_i votes for d_c only if p_i has received an rQC or an sQC for d and $d_c.\text{stable} = d$. \square

Lemma B.2. Suppose that qc_b and qc_d are two qualified QCs, and each is received by at least one correct replica. Let b and d be $\text{QCBLOCK}(qc_b)$ and $\text{QCBLOCK}(qc_d)$, respectively. If b and d are two conflicting blocks, then $\text{rank}(b) \neq \text{rank}(d)$.

Proof. Assume, on the contrary, that $\text{rank}(b) = \text{rank}(d)$. Let v denote the view of b and d . As each qualified QC consists of at least $2f + 1$ votes, at least one correct replica has voted for both b and d . Let b' and d' denote the parent block of b and d , respectively. Since a correct replica votes for at most one block of each height during normal case operation, at least one of b and d is proposed during view change. Therefore, $b'.\text{view} < v$ or $d'.\text{view} < v$. Now we consider two cases:

(1) $b'.\text{view} < v$ and $d'.\text{view} < v$. According to Algorithm 7, a correct replica p_i votes for at most one block that extends a block proposed in a lower view. Hence, b equals d .

(2) ($b'.\text{view} < v$ and $d'.\text{view} = v$) or ($b'.\text{view} = v$ and $d'.\text{view} < v$). If $b'.\text{view} < v$ and $d'.\text{view} = v$, then there exists a block d_0 of lowest height on the branch led by d

such that $d_0.\text{view} = v$. Hence, the view of $d_0.\text{parent}$ is lower than v . Let d'_0 denote a block on the branch led by d such that $d'_0.\text{parent} = d_0$. By Lemma B.1, at least one correct replica p_i has voted for d'_0 . According to ln 16-25 in Algorithm 6, p_i has received an rQC or an sQC for d_0 . Note that the view of $d_0.\text{parent}$ is lower than v . Then d_0 and b must be the same block according to case (1). Therefore, d is an extension of b . The proof is similar in the case where $b'.\text{view} = v$ and $d'.\text{view} < v$.

In both cases, d and b are either the same block or on the same branch, contradicting the condition that they are conflicting blocks. Therefore, $\text{rank}(b) \neq \text{rank}(d)$. \square

Lemma B.3. If a correct replica has voted for d and set its vb to d , then d must be an extension of $d.\text{stable}$ and at least one correct replica has received a qualified QC for $d.\text{stable}$.

Proof. Let d_0 denote $d.\text{parent}$. Let p_i denote a correct replica that has voted for d and set its vb to d . According to ln 19-25 of Algorithm 6, p_i has received a certificate qc for d_0 , where qc and d are matching. We distinguish two cases.

(1) qc is an rQC or an sQC for d_0 and $d.\text{stable} = d_0$ (ln 23-25 in Algorithm 6). In this case, d is an extension of $d.\text{stable}$ and p_i received a qualified QC for $d.\text{stable}$.

(2) qc is a wQC for d_0 and $d.\text{stable} = d_0.\text{stable}$ (ln 19-22 in Algorithm 6). Let d_v denote the block of lowest height on the branch led by d such that $d_v.\text{stable} = d.\text{stable}$. Let d'_v denote $d_v.\text{parent}$. Then $d_v.\text{stable} \neq d'_v.\text{stable}$. According to Lemma B.1, at least one correct replica p_j has voted for d_v since $d_v.\text{stable} \neq d'_v.\text{stable}$. Note p_j votes for d_v only if one of the following conditions holds: i) $d_v.\text{stable} = d'_v.\text{stable}$; ii) $d_v.\text{stable} = d'_v$ and p_j receives a qualified QC for d'_v . In this case, $d_v.\text{stable} = d.\text{stable} = d'_v$, d is an extension of $d.\text{stable}$, and p_j has received a qualified QC for $d.\text{stable}$.

Either way, d is an extension of $d.\text{stable}$ and at least one correct replica has received a qualified QC for $d.\text{stable}$. \square

Lemma B.4. If a qualified QC is formed in view v , then there exists only one block b of lowest rank for which a qualified QC is formed in view v , and we have that:

- (1) the view of $b.\text{parent}$ is lower than v ;
- (2) if there exists a qualified QC for b_1 , $b_1.\text{view} = v$, and the view of $b_1.\text{parent}$ is lower than v , then b_1 equals b ;
- (3) if there exists a qualified QC for d and $d.\text{view} = v$, then d equals b or d is an extension of b .

Proof. If a qualified QC is formed in view v , then there exists only one block b of lowest rank for which a qualified QC is formed in view v (according to Lemma B.2).

(1) Let b_v denote the block of lowest height such that $b_v.\text{view} = v$ on the branch led by b . We have $b_v.\text{height} \leq b.\text{height}$ and the view of $b_v.\text{parent}$ is lower than v . If $b_v \neq b$, then there exists a block b'_v on the branch led by b such that $b'_v.\text{parent} = b_v$ and $b'_v.\text{view} = b_v.\text{view} = v$. From Lemma B.1, at least one correct replica p_i has received an rQC or an sQC for b_v . Thus, b_v is a block of lower rank than b and a qualified

QC for b_v is formed in view v , contradicting the definition of b . Hence, we have $b_v = b$ and the view of $b.parent$ is lower than v .

(2) If there exists a qualified QC for b_1 , at least one correct replica has voted for both b_1 and b in view v . According to Algorithm 7, in view v , a correct replica only votes for one block that extends a block proposed in a lower view than v . Therefore, it must hold that $b_1 = b$.

(3) There exists a qualified QC for d and $d.view = v$. Let d_0 denote the block of lowest height on the branch led by d such that $d_0.view = v$. Then the view of the parent block of d_0 is lower than v . From Lemma B.1, a correct replica has received a qualified QC for d_0 . According to claim (2), we know d_0 equals b . Therefore, d equals b or d is an extension of b . \square

Lemma B.5. *For any qualified QC qc , if $QCBlock(qc) = b$ and $b.view = v$, then any block proposed in view v on the branch led by b has been voted for by at least $f + 1$ correct replicas.*

Proof. Assume that block d is on the branch led by b such that $d.view = v$ and fewer than $f + 1$ correct replicas have voted for d . We immediately know that a qualified QC for d cannot be formed. Let d' denote a block such that $d'.parent = d$. So, a correct replica p_i votes for d' only if a wQC for d is received and p_i has voted for d . Since fewer than $f + 1$ correct replicas have voted for d , a qualified QC for d or any extensions of d (including b) cannot be formed (a contradiction). \square

Lemma B.6. *For any two qualified QCs qc_1 and qc_2 , let b_1 and b_2 be $QCBlock(qc_1)$ and $QCBlock(qc_2)$, respectively. If b_1 is conflicting with b_2 , then $b_1.view \neq b_2.view$.*

Proof. Assume, on the contrary, that $b_1.view = b_2.view = v$. Let b be the block of lowest height for which a qualified QC was formed in view v . Then according to Lemma B.4, either b_1 or b_2 equals b or is an extension of b . Hence, $b_1.height \geq b.height$ and $b_2.height \geq b.height$. We consider three cases:

(1) If $b_1.height = b.height$ or $b_2.height = b.height$, then b_1 equals b or b_2 equals b . Therefore, b_1 and b_2 are the same block or they are on the same branch.

(2) If $b.height < b_1.height$, $b.height < b_2.height$, and $b_1.height = b_2.height$, then according to Lemma B.2, b_1 and b_2 must be the same block.

(3) If $b.height < b_1.height$, $b.height < b_2.height$, and $b_1.height \neq b_2.height$, then b_1 and b_2 are extensions of b . W.l.o.g., we assume that $b_1.height < b_2.height$. Let b'_2 denote a block on the branch led by b_2 such that $b'_2.height = b_1.height$. Then b'_2 is an extension of b and b'_2 and b_1 are blocks proposed during the normal case operation in view v . According to Lemma B.5, at least $f + 1$ correct replicas have voted for b'_2 . Since each rQC consists of at least $2f + 1$ votes, at least one correct replica has voted for both b'_2 and b_1 . Note that during the normal case operation, a correct replica votes for at most one block of each height. Therefore, it holds that

b'_2 and b_1 must be either the same block or on the same branch.

In all cases, b_1 and b_2 are the same block or are blocks on the same branch, contradicting the condition that they are conflicting blocks. Therefore, $b_1.view \neq b_2.view$. \square

Lemma B.7. *Suppose that all the correct replicas have voted for b in view v , $b.parent = b.stable$ and $b.parent$ is proposed in view v . If a correct replica has received a wQC qc for d such that $rank(d.stable) \geq rank(b.parent)$, and d , $d.parent$, and $d.stable$ are blocks proposed in view v , then d equals b or d is an extension of b .*

Proof. As b , $b.parent$, d , and $d.parent$ are all blocks proposed in view v , b and d are blocks proposed during normal case operation in view v . According to Algorithm 6, we know that if a correct replica has voted for d , the replica will set its vb to d at the same time. Since qc consists of $f + 1$ votes, at least one correct replica has voted for d . From Lemma B.3, d is an extension of $d.stable$ and at least one correct replica has received a qualified QC for $d.stable$. Now we consider two cases:

(1) $rank(d.stable) = rank(b.parent)$. Since $b.parent = b.stable$, any correct replica votes for b only after receiving a qualified QC for $b.parent$. Then $d.stable = b.parent$ and $d.height \geq b.height$ (according to Lemma B.2). Let d' denote the block on the branch led by d such that $d'.height = b.height$. Then at least one correct replica has voted for d' in view v according to Lemma B.1. Since correct replicas vote for at most one block of each height during normal operation in a view, d' must be equal to b . Therefore, d equals b or d is an extension of b .

(2) $rank(d.stable) > rank(b.parent)$. It is straightforward to see that $rank(d.stable) \geq rank(b)$. According to Lemma B.6, $d.stable$ is equal to b or $d.stable$ is an extension of b . Hence, d is an extension of b . \square

Lemma B.8. *For a commitQC qc for b and a qualified QC qc_d for d , if $rank(b) < rank(d)$, then d must be an extension of b .*

Proof. Let v denote $b.view$ and v_d denote $d.view$. As $rank(d) > rank(b)$, then $v_d \geq v$. Let b' denote $QCBlock(qc)$. Since qc is a commitQC for b , there are two conditions: (1) qc is an rQC for b' , $b'.stable = b'.parent = b$ and $b'.view = v$; (2) qc is an sQC for b , $b.parent = b.stable$ and the view of $b.parent$ equals v .

We prove the lemma by induction over the view v_d , starting from view v .

Base case: Suppose $v_d = v$. From Lemma B.6, for condition (1) or (2), d must be an extension of b .

Inductive case: Assume this property holds for view v_d from v to $v + k - 1$ for some $k \geq 1$. We now prove that it holds for $v_d = v + k$.

Let d_0 denote the block of lowest height on the branch led by d such that $d_0.view = v_d$. Then the view of the parent

block of d_0 is lower than v_d , d_0 is proposed during view change in view v_d , and d_0 is voted for by at least one correct replica p_i (Lemma B.1).

Let m denote the `NEW-VIEW` message for d_0 . According to Algorithm 7, $m.justify$ is a set M of $2f + 1$ `VIEW-CHANGE` messages for view v_d . Let qc_1 denote the qualified QC with the highest rank contained in $M.justify$ and let b_1 denote `QCBlock`(qc_1). For all the wQCs contained in $M.justify$, a correct replica chooses the wQC for a block with the highest stable block according to Ln 19-24 in Algorithm 5 and sets the wQC as vc . Let b_0 denote `QCBlock`(vc). Note that b_0 , $b_0.parent$ and $b_0.stable$ are proposed in the same view. Then b_0 is a block proposed during the normal case operation. Let b_2 denote the block which is included in more than $f + 1$ messages in M . If no such block exists, b_2 is set to \perp .

In view v_d , p_i votes for d_0 if $d'_0 = d_0.parent$, $d'_0.view < v_d$, $d'_0.height + 1 = d_0.height$ and one of the following conditions are satisfied:

- i) $d'_0 = b_2$, $rank(b_2.stable) \geq rank(b_1)$ (Ln 27 in Algorithm 5).
- ii) $d'_0 = b_0$, i) is not satisfied and $rank(b_0.stable) \geq rank(b_1)$ (Ln 28 in Algorithm 5).
- iii) $d'_0 = b_1$, i) and ii) are not satisfied (Ln 29 in Algorithm 5).

Note that b_0 is a block proposed during the normal case operation in view $b_0.view$. Since a wQC consists of $f + 1$ votes, at least one is sent by a correct replica. Hence, at least one correct replica has voted for b_0 and sets its vb as b_0 . According to Lemma B.3, b_0 is an extension of $b_0.stable$ and at least one correct replica has received a qualified QC for $b_0.stable$.

Next, we prove the property holds in view $v + k$ for the two situations for `commitQC`, respectively.

(1) qc is an rQC. Let S denote the set of correct replicas that have received a qualified QC for b in view v . Since in view v correct replicas vote for b' only after receiving a qualified QC for b , we have $|S| \geq f + 1$. Note that a correct replica updates its QC_r only with a qualified QC with a higher rank. Thus, for any `VIEW-CHANGE` message sent by a replica in S , the `justify` field is set to a qualified QC with the same or a higher rank than b . Since M consists of $2f + 1$ messages, at least one message in M is sent by a replica in S . Therefore, $rank(b_1) \geq rank(b)$ and $b_1.view < v_d$.

According to the inductive hypothesis, b_1 must be equal to b or an extension of b . Therefore, if condition iii) is satisfied, d_0 must be an extension of b . If condition i) is satisfied, then $rank(b_2) > rank(b_1)$ and $rank(b_2.stable) \geq rank(b_1)$. Since at least one correct replica has set its vb to b_2 , then b_2 is an extension of $b_2.stable$ and a qualified QC qc_2 for $b_2.stable$ has been received by a correct replica from Lemma B.3. According to the inductive hypothesis, b_2 is an extension of b . Hence, d'_0 is an extension of b . If condition ii) is satisfied, then $rank(b_0.stable) \geq rank(b_1)$. Note that b_0 is an extension of $b_0.stable$ and at least one correct replica has received a qualified QC for $b_0.stable$. Thus, b_0 is an extension of b

(according to the inductive hypothesis). Therefore, d'_0 is an extension of b . No matter which condition is satisfied, both d_0 and d must be extensions of d'_0 and extensions of b .

(2) qc is an sQC, the view of $b.parent$ equals v and $b.parent = b.stable$. Since qc consists of $3f + 1$ votes, all the correct replicas have received a qualified QC for $b.parent$, changed its QC_r to a qualified QC for $b.parent$, and voted for b in view v . Let V denote the set of correct senders of messages in M . It is clear that $|V| \geq f + 1$. Since correct replicas only change their QC_r to a qualified QC of a higher rank, we have $rank(b_1) \geq rank(b.parent)$.

(a) If $rank(b_1) \geq rank(b)$, then from Lemma B.2 and the induction hypothesis, b_1 is equal to b or b_1 is an extension of b . If condition iii) is satisfied, then d_0 and d are extensions of b . If condition i) or ii) is satisfied, at least one correct replica has voted for d'_0 and set its vb to d'_0 , and $rank(d'_0.stable) \geq rank(b_1)$. According to Lemma B.3, d'_0 is an extension of $d'_0.stable$ and at least one correct replica has received a qualified QC for $d'_0.stable$. Again, from the induction hypothesis, $d'_0.stable$ is equal to b or $d'_0.stable$ is an extension of b . Therefore, d_0 and d are extensions of b .

(b) If $rank(b_1) < rank(b)$, then $rank(b_1) = rank(b.parent)$. If $b_2 = b$, then condition i) is satisfied. Hence, d'_0 equals b and d_0 and d are extensions of b .

If $b_2 \neq b$, then there exists a correct replica p_i in V such that when p_i sent a `VIEW-CHANGE` message for v_d , its last voted block vb is b_e and $b_e \neq b$. Let b'_e denote $b_e.parent$. According to Ln 19-22 in Algorithm 6, p_i has received a wQC qc_e for b'_e , $rank(b'_e) \geq rank(b)$, and $rank(b'_e) \geq rank(b.parent)$. If $b'_e.view = v$, then b'_e equals b or b'_e is an extension of b from Lemma B.7. If $b'_e.view > v$, then the view of $b'_e.stable$ is higher than v . From Lemma B.3, b'_e is an extension of $b'_e.stable$ and a correct replica has received a qualified QC for $b'_e.stable$. From the inductive hypothesis, as $rank(b'_e.stable) > rank(b)$, it must hold that $b'_e.stable$ is an extension of b . Therefore, b_e must be an extension of b , b_2 is set to \perp or b_2 is an extension of b . If condition i) is satisfied, d'_0 equals b_2 . We know that p_i has sent qc_e in its `VIEW-CHANGE` message. Then $rank(b_1.stable) \geq rank(b.parent)$. If condition i) is not satisfied, condition ii) is satisfied and d'_0 equals b_1 . Note that a wQC for b_1 is included in M and b_1 is proposed during normal case operation. Similar to b'_e , b_1 must be an extension of b . Either way, d'_0 is equal to or an extension of b . Thus, d_0 and d are extensions of b .

Therefore, d must be an extension of b and the property holds in view $v + k$ based on Case (1) and Case (2). This completes the proof of the lemma. \square

Theorem B.9. (safety) *If b and d are conflicting blocks, then not both can be committed by at least one correct replica.*

Proof. Suppose that there exist `commitQC`'s for both b and d . According to Lemma B.1, a qualified QC must have been formed for both b and d . From Lemma B.2, if $rank(b) = rank(d)$, only one qualified QC for b and d can be formed in

the same view. For the case where $\text{rank}(b) \neq \text{rank}(d)$, we assume w.l.o.g. that $\text{rank}(b) < \text{rank}(d)$. From Lemma A.7, we know that a qualified QC for d cannot be formed in view $d.\text{view}$. This completes the proof of the theorem. \square

Theorem B.10. (liveness) *After GST, there exists a bounded time period T_f such that if the leader of view v is correct and all correct replicas remain in view v during T_f , then a decision is reached.*

Proof. Suppose after GST, in a new view v , the leader p_i is correct. Then p_i can collect a set M of $2f + 1$ VIEW-CHANGE messages from correct replicas and broadcast a new block b_v in a NEW-VIEW message m . Since $m.\text{justify}$ contains M , every correct replica can verify the block b_v using a call `SAFEBLOCK(M)`.

Under the assumption that all correct replicas are synchronized in their view, p_i is able to form a QC for b and generate new blocks. All correct replicas will vote for the new blocks from p_i . Therefore a *commitQC* for b can be formed by p_i and any correct replica will vote for b . After GST, the duration T_f for these phases to complete is of bounded length. \square

C The Underlying BFT Protocol in Star

C.1 The Consensus Protocol Implemented in Star

We now describe the concrete atomic broadcast protocol that we implemented in Star. We use a variant of PBFT that differs from PBFT in two minor aspects. The protocol we will describe in the following is not presented in its general manner but instead takes as input the output from the transmission process.

Normal case operation. We first describe the normal case protocol.

Step 1: Pre-prepare. The leader checks whether $|W[le]| \geq n - f$. If so, it proposes a block B and broadcasts a `(PRE-PREPARE, v, B)` message to all replicas.

The block B is of the form $\langle v, \text{cmd}, \text{height} \rangle$, where v is the current view number, $B.\text{cmd} = W[le]$, and $B.\text{height} = le$. We directly use $B.\text{height}$ as the sequence number for B in the protocol.

Step 2: Prepare. Replica receives a valid PRE-PREPARE message for block B and broadcasts a PREPARE message.

After receiving a PRE-PREPARE message `(PRE-PREPARE, v, B)` from the leader, a replica p_j first verifies whether 1) its current view is v , 2) $B.\text{cmd}$ consists of at least $n - f$ wQCs or rQCs for epoch e , and 3) p_j has not voted for a block $B.\text{height}$ in the current view. Then p_j broadcasts a signed PREPARE message `(PREPARE, $v, \text{hash}(B)$)`. The replica also updates its W queue if any QC included in $B.\text{cmd}$ is not in $W[B.\text{height}]$.

Step 3: Commit. Replica receives $n - f$ PREPARE messages for B and broadcasts a COMMIT message.

After receiving $n - f$ matching PREPARE messages with the same $\text{hash}(B)$, replica p_j combines the messages into a regular certificate for B , called a *prepare certificate*. Then p_j

broadcasts a `(COMMIT, $v, \text{hash}(B)$)` message. After receiving $n - f$ COMMIT messages with the same $\text{hash}(B)$, p_j *a-delivers* B with sequence number le .

Note that the PRE-PREPARE step and the COMMIT step carry only $\text{hash}(B)$ as the message transmitted. The total communication for the normal case operation is thus $O(n^2\lambda)$ where λ is the security parameter.

Checkpointing. After a fixed number of blocks are a-delivered, replicas execute the checkpoint protocol for the garbage collection. Each replica broadcasts a checkpoint message that includes its current system state and the epoch number for the latest a-delivered block. Each replica waits for $n - f$ matching checkpoint messages which form a stable checkpoint. Then the system logs for epoch numbers lower than the stable checkpoint can be deleted.

View change. We now describe the view change protocol. After a correct replica times out, it sends a VIEW-CHANGE message to all replicas. Upon receiving $f + 1$ VIEW-CHANGE messages, a replica also broadcasts a VIEW-CHANGE message. The new leader waits for $n - f$ VIEW-CHANGE messages, denoted as M , and then broadcasts a NEW-VIEW message to all replicas.

The VIEW-CHANGE message is of the form `(VIEW-CHANGE, C, \mathcal{P})`, where C a stable checkpoint and \mathcal{P} is a set of prepare certificates. For \mathcal{P} , a prepare certificate certificate for each epoch number greater than C and lower than the replica's last vote is included.

The NEW-VIEW message is of the form `(NEW-VIEW, $v + 1, c, M, \mathcal{PP}$)`, where c is the latest stable checkpoint, M is the set of VIEW-CHANGE messages M , and \mathcal{PP} is a set of PRE-PREPARE messages. The set \mathcal{PP} is computed as follows: For each epoch number e between C and the epoch number of any replica's last vote, the new leader creates a new PRE-PREPARE message. If a prepare certificate is provided by any replica in the VIEW-CHANGE message, the PRE-PREPARE message is of the form `(PRE-PREPARE, $v + 1, h$)`, where h is the hash in the prepare certificate. If none of the replicas provides a prepare certificate, the new leader creates a message `(PRE-PREPARE, $v + 1, B$)`, where B is of the form $\langle v + 1, W[e], e \rangle$.

Upon receiving a NEW-VIEW message, a replica verifies the PRE-PREPARE messages in the \mathcal{PP} field by executing the same procedures as the leader based on M . Then the replicas resume normal operation.

D Correctness of Star

Based on the safety and liveness properties of the underlying atomic broadcast protocol in the consensus process, we now prove the correctness of Star.

According to the Star specification, a set V consisting of transactions in batches $\{\text{QCPROPOSAL}(qc_k)\}_{k \in [1..n-f]}$ delivered (in a deterministic order) by p_i must correspond to the set m (consisting of $n - f$ wQCs $\{qc_k\}_{k \in [1..n-f]}$) a-delivered

by p_i from the underlying atomic broadcast protocol. In this case, we simply say V is *associated with* m .

We prove the safety of Star by showing that different sets of transactions cannot be committed together in the same epoch, each by a correct replica. We begin with the following lemma:

Lemma D.1. *If V_i associated with some m is delivered by p_i and V_j associated with the same m is delivered by p_j , then we have $V_i = V_j$.*

Proof. Assume, towards contradiction, that $V_i \neq V_j$. Let $\{qc_k\}_{k \in [1..n-f]}$ be the $n - f$ wQCs contained in m . Then we have that V_i is a union of transactions in proposals $\{b_k\}_{i \in [1..n-f]}$, where we have $b_k = \text{QCPROPOSAL}(qc_k)$. Similarly, V_j is a union of transactions in proposals $\{b'_k\}_{i \in [1..n-f]}$, where $b'_k = \text{QCPROPOSAL}(qc_k)$. Since $V_i \neq V_j$, we have that there exists $k \in [1..n - f]$ such that $b_k \neq b'_k$. Note that qc_k is a wQC for b_k and also a wQC for b'_k . Since $b_k \neq b'_k$, this violates the unforgeability of digital signatures (or threshold signatures). \square

Now we are ready to prove safety.

Theorem D.2. (safety) *If a correct replica delivers a transaction tx before delivering tx' , then no correct replica delivers a transaction tx' without first delivering tx .*

Proof. Suppose that a correct replica p_i delivers a transaction tx before delivering tx' . Let L_i denote the a-delivered messages log of p_i and TL_i denote the delivered transactions log of p_i . For any correct replica p_j , let L_j denote the a-delivered messages log and TL_j denote the delivered transactions log of p_j . According to the safety of the consensus protocol, either L_i equals L_j or one of L_i and L_j is a prefix of the other. Note that TL_i and TL_j contain transactions associated with messages in the a-delivered messages logs in a deterministic order. According to Lemma D.1, either TL_i equals TL_j or one of TL_i and TL_j is a prefix of the other. This completes the proof of the theorem. \square

Theorem D.3. (liveness) *If a transaction tx is submitted to all correct replicas, then all correct replicas eventually deliver tx .*

Proof. If a transaction tx is submitted to all correct replicas, eventually in some epoch, tx is included in the proposal by at least one correct replica. Using the strategy in EPIC (following HoneyBadgerBFT), eventually the wQC wqc for the proposal containing the transaction tx will be sent to the consensus process.

At least $n - f$ wQCs will be a-delivered in the consensus process, and at least $f + 1$ wQCs must be proposed by correct replicas. So there is some probability that wqc for tx will be delivered. If the corresponding transaction has been received by a correct replica, then we are done. Otherwise, a correct replica just needs to run the fetch operation to get the corresponding proposal containing tx . Recall that

the use of wQC ensures that a correct replica must have stored the corresponding proposal. (If the underlying atomic broadcast only achieves consistency rather than agreement, then we can still use the standard state machine replication mechanism such as state transfer to ensure that all correct replicas deliver the transaction.) \square

E Additional Related Work

Much related work is discussed in the course of the paper. Here we discuss additional related work.

HotStuff and its derivatives. HotStuff [52] is known as the first partially synchronous BFT protocol with linearity. HotStuff has three round-trips for both normal case operations and view changes. Subsequent works focus on reducing the number of phases for HotStuff, including Fast-HotStuff [29], Jolteon [23], Wendy [24], and Marlin [49]. Kauri is a recent variant that targets for scalability [44]. HotStuff has a basic mode and a chained (pipelining) mode (called chained HotStuff). The protocols introduced and implemented in this paper are described in their chained mode.

Protocol switching in BFT protocols. Following the idea initially proposed by Kursawe and Shoup [34], Bolt-Dumbo [38] and Ditto [23] run partially synchronous BFT protocols in the optimistic mode and rely on fallback asynchronous protocols during asynchrony [34]. All known such BFT protocols lack an effective mechanism to decide when to switch from the pessimistic mode to the optimistic mode, as it is (often) unpredictable when the network becomes synchronous again. The situation is only exacerbated, if the network is intermittently synchronous or adaptively manipulated [42]. Meanwhile, systems have been proposed to allow switching among (restricted) BFT protocols [6, 27]. These protocols offer adaptive performance but (largely) inherit the issues of protocol switching [23, 34, 38].

Weak certificates for eventual consistency. Zeno[45] uses weak certificates to handle network partitions. It allows $f + 1$ replicas to make progress, including view changes. Zeno leverages a conflict-resolution mechanism to achieve eventual synchrony, a consistency goal that is much weaker than ours. Several protocols utilize trusted subsystem to build BFT that requires $2f + 1$ replicas to tolerate f Byzantine failures [17, 18, 31]. In such protocols, the quorum size is $f + 1$. In contrast, Dashing1 combines weak certificates and regular certificates to achieve standard BFT guarantees, additionally gaining in efficiency and robustness.

BFT with strong quorums. Strong quorums (with $3f + 1$ replicas) for consensus have been used in Zyzzyva [33] and FaB [41]. The protocols have been found to have errors [3] which were fixed [4]. The fixed algorithm is at the center of SBFT [26] which also features the usage of strong quorums. This concept is also related to speculative execution [33, 40] that enjoys a fast path for the consensus. Dashing2 tackles

the new and subtle challenges due to weak certificates and linear communication.

Multiple thresholds in a single timing model or two different timing models. Some Byzantine-resilient protocols such as UpRight [15, 28] study different thresholds for different correctness properties (e.g., different thresholds for safety and liveness) in a single timing model. Some other protocols, however, consider two different timing models. Most of these protocols (except XFT [37]) focus on the asynchronous-synchronous timing model [8, 9, 30, 39, 43, 51]. For instance, the recent work of Malkhi, Nayak, and Ren [39] and the work of Momose and Ren [43] consider these two timing models and separate thresholds for safety and liveness properties. In contrast, XFT considers the partially synchronous-synchronous timing model. XFT tolerates $f < n/2$ Byzantine failures under synchrony but no Byzantine failures under partial synchrony.

Our protocols are different from these protocols. Our protocols use a single timing model and assume the $n/3$ threshold for both safety and liveness. The different thresholds in our protocols are used to improve efficiency and robustness. **Tusk.** It is worth noting that Tusk (an asynchronous protocol) may appear to have slightly better performance than Narwhal (a partially synchronous protocol) [1]. We emphasize that this is mainly because Tusk does not implement the underlying threshold common coin protocol, a key performance bottleneck; instead, Tusk uses epoch numbers as common coins. Also recall that both Tusk and Narwhal only achieves a weak liveness property.