Byzantine Fault Tolerance from Weak Certificates

Sisi Duan
Tsinghua University
duansisi@tsinghua.edu.cn

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

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

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

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

Gang Di
Digital Currency Institute, the
People's Bank of China
digang@pbcdci.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 in this paper that one can use *weak certificates* obtained from only $f+1$ signatures to design more robust and much more efficient BFT protocols. We devise Dashing (a family of three HotStuff-style BFT protocols) and Star (a parallel BFT framework).

We begin with Dashing1 that targets both efficiency and robustness using weak certificates. Dashing1 is partition-tolerant and network-adaptive, and does not rely on fallback asynchronous BFT protocols. Dashing2 is a variant of Dashing1 and focuses on performance only. Then we show in Dashing3 how to further enable a fast path by using *strong certificates* obtained from $3f+1$ signatures, a challenging task we tackled in the paper.

We then leverage weak certificates to build a highly efficient BFT framework (Star) that delivers transactions from $n-f$ replicas using only a single consensus instance in the standard BFT model. Star completely separates bulk data transmission from consensus. Moreover, its data transmission process uses $O(n^2)$ messages only and can be effectively pipelined.

We demonstrate that the Dashing protocols achieve 10.7%-29.9% higher peak throughput than HotStuff. Meanwhile, Star, when being instantiated using PBFT, is an order of magnitude faster than HotStuff. Furthermore, unlike the Dashing protocols and HotStuff whose performance degrades as f grows, the peak throughput of Star increases as f grows. When deployed in a WAN with 91 replicas across five continents, Star achieves 243 ktx/sec, 15.8x the throughput of HotStuff.

1 INTRODUCTION

Byzantine fault-tolerant state machine replication (BFT) is known as the core building block for permissioned blockchains. This paper focuses on highly efficient, partially synchronous BFT protocols [15, 19]. 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 [22] and HotStuff [41] require using (threshold) signatures for regular certificates as transferable proofs.

This paper demonstrates that one can build various BFT protocols that outperform existing ones (in one way or another) by leveraging *weak certificates* which are sets 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 a recent paper [18], asynchronous BFT protocols using common coins 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, Dashing3, and Dashing3) are new BFT protocols in the HotStuff family, while Star is a new (asynchronous) BFT framework that has $O(n^2)$ messages, separating message transmission from consensus, and allowing pipelining in message transmission. Below, let us describe the features of protocols.

1.1 The Dashing Protocols: Gaining in Robustness and Efficiency

Dashing1 is built on top of the HotStuff protocol and designed with two goals in mind: 1) efficiency (due to the usage of weak certificates) and 2) robustness (again due to the usage of weak certificates).

On the one hand, it is known that HotStuff and all its derivatives [20, 21, 25] (have to) use regular certificates to ensure liveness and safety. It is an interesting open problem to provide a meaningful way of using weak certificates in BFT protocols for better efficiency.

On the other hand, partially synchronous BFT protocols cannot make progress during asynchrony (e.g., network partitions) and may loop on view changes (leader election). Recent HotStuff-style protocols, such as Ditto [20] and Bolt-Dumbo [31], run BFT protocols in the optimistic mode and rely on fallback asynchronous protocols during asynchrony [28]. When the protocols are in the pessimistic mode, their performance is significantly reduced. Moreover, 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. The situation is only exacerbated, if the network is intermittently synchronous or adaptively manipulated [34]. It is

protocols	described in	QC type used	features	authenticator	communication
Dashing1	Sec. 4.4	wQC; rQC	partition-tolerant; more robust and efficient	$O(n)$	$O(Ln + \lambda n)$
Dashing2	Sec. 4.5	wQC; rQC	derived from Dashing1; targeting performance only	$O(n)$	$O(Ln + \lambda n)$
Dashing3	Sec. 4.6	wQC; rQC; sQC	targeting low latency; one-phase fast path	$O(n)/O(n^2)$	$O(Ln + \lambda n^2)$
Star	Sec. 5	wQC; rQC	(separating bulk data transmission from consensus;) supporting parallelism using one consensus instance; pipelined data transmission; lower message and communication complexity	$O(n^2)$	$O(Ln^2 + \lambda n^2)$

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

natural to ask if we have alternative solutions to build network-adaptive and efficient BFT protocols.

Dashing1. The idea underlying Dashing1 is that when the network partition occurs, we allow replicas to proceed with weak certificates. Once the network becomes synchronous and a transaction with a regular certificate is delivered, *all* prior transactions with weak certificates are immediately delivered. The mechanism is partition-tolerant, as the system can make meaningful progress in the presence of partitions, and network adaptive in the sense that as long as the leader can collect weak certificates from $f + 1$ replicas, progress will be eventually made. Dashing1 may avoid unnecessary view changes and accelerate transaction processing in both failure-free and failure scenarios.

Turning the idea into a fully-fledged BFT protocol, however, is highly challenging. We demonstrate the safety and liveness issues and provide an efficient protocol for handling them.

Dashing2. Dashing2 is a (simple) variant of Dashing1 and *targets performance only*. In particular, we use two ideas to improve the system throughput. First, we adopt the idea of using $f + 1$ actively participating replicas for normal case operations [26]. Second, we ask replicas to periodically switch between weak certificates and regular certificates to guarantee that transactions with weak certificates are committed in a timely manner. For instance, replicas can process one transaction using a regular certificate immediately after ten transactions using weak certificates.

Dashing3. 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 and provide an efficient and secure solution.

1.2 Star: Gaining in Parallelism and Pipelining

We use weak certificates to build a partially synchronous BFT protocol (Star) that can deliver transactions from $n - f$ replicas using only a single consensus instance. Star assumes the standard BFT model with no other supporting servers.

As depicted in Fig. 1, Star completely separates bulk data transmission from consensus such that these two processes can be run independently. In particular, the data transmission process can be effectively *pipelined*, which significantly improves the performance.

Star can be viewed as a BFT protocol that inherits the benefits of two beautiful recent systems—Narwhal [17] and ISS [38], and further improves on them by using new techniques.

As in Narwhal, Star dissociates data dissemination from consensus. Star, however, works in the standard BFT model and does not rely on additional nodes helping data transmission; more crucially,

with our new design using weak certificates, the pipelining and parallelism features of the data transmission process is no longer the obvious performance bottleneck.

Similar to ISS (a partially synchronous BFT framework), Star can deliver transactions from $n - f$ replicas. Despite the similarity, the idea underlying our design is actually motivated by completely asynchronous BFT protocols [8, 14, 17]. As a result, different from ISS that requires running n consensus protocols, Star only needs to run a single consensus protocol for each epoch. Also, 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, the message complexity of Star is only $O(n^2)$ even in the worst case. A detailed comparison can be found in Sec. 2.
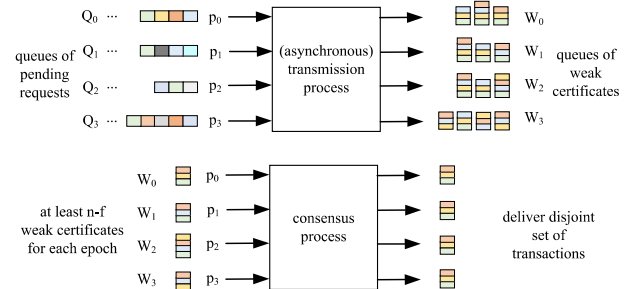

Figure 1: The Star BFT framework. Star consists of an asynchronous transmission process (that takes as input queues of pending transactions and outputs queues of weak certificates) and a consensus process (that takes at input $n - f$ weak certificates and outputs a union of transactions corresponding to the weak certificates delivered).

1.3 Summary of Contributions

We summarize our contributions in the following.

- We design a family of Dashing protocols (Dashing1, Dashing2, and Dashing3) using weak certificates. Compared to prior protocols, our protocols offer improved performance and robustness. Besides, they also have their unique features and provide interesting trade-offs. We have added novel techniques to the HotStuff family.

- We provide a novel (asynchronous) BFT framework allowing one to process transactions in parallel using only one BFT instance and $O(n^2)$ messages. We also give an efficient instantiation based on PBFT.
- We formally prove the correctness of all our protocols.
- We implement the four new BFT protocols (the three Dashing protocols and an instantiation of Star). We have performed extensive evaluation of the protocols and we show that these protocols outperform existing protocols. We briefly summarize the evaluation results in the following:
  (1) The peak throughput of Dashing1 is 21.5% higher than that of HotStuff for $f = 1$. When $f = 30$, Dashing1 still achieves 28.1% higher peak throughput than HotStuff.
  (2) The peak throughput of Dashing2 is 2.6%-9.8% higher than that of Dashing1. The throughput of Dashing1 and Dashing3 is very close, but Dashing3 in general achieves lower latency by using strong certificates, at the cost of more expensive view changes.
  (3) The peak throughput of Star is an order of magnitude higher than Dashing protocols and HotStuff in most scenarios. Unlike Dashing protocols and HotStuff whose performance degrades as f grows, the peak throughput of Star increases as f grows. When $f = 30$ in WAN environments, Star achieves 15.8x the throughput of HotStuff.

2 RELATED WORK

HotStuff and its derivatives. HotStuff [41] 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 [25], Jolteon [20], and Wendy [21].

HotStuff has a basic mode and a chained (pipelining) mode (called chained HotStuff). The protocols presented and introduced in this paper are described in their chained mode.

The technique underlying HotStuff has also been shown significant in building various Byzantine-resilient protocols [4–6, 40].

Protocol switching in BFT protocols. Following the idea initially proposed by Kursawe and Shoup [28], Bolt-Dumbo [31] and Ditto [20] are two recent systems that provide high performance in the steady state by using partially synchronous protocols and provide progress during asynchrony by using asynchronous ones.

Our protocol Dashing1 provides an alternative solution to address network asynchrony. On the one hand, Dashing1 tolerates many forms of asynchrony, e.g., network partition. As long as there are $f + 1$ correct replicas in the partition with the leader, Dashing1 may make meaningful progress. Also, Dashing1 also works well in the intermittently asynchronous environment where the network often experience interruptions [34]. On the other hand, Dashing1 is network-adaptive, in the sense that once the network becomes synchronous or less partitioned, Dashing1 can make faster progress and commit all prior pending transactions (with weak certificates) immediately. But of course, Dashing1, being a partially synchronous BFT protocol, does not tolerate the general asynchronous network scheduler.

Meanwhile, systems and frameworks [7, 23] have been proposed to allow switching among different partially synchronous BFT protocols. These protocols offer excellent adaptive performance but do not tolerate network asynchrony: during asynchrony, they may achieve zero throughput [34].

BFT with strong quorums. Strong quorums (with $3f + 1$ replicas) for consensus have been used in Zyzzyva [27] and FaB [33]. The protocols have been found to have errors [2] and then fixed [3]. The fixed algorithm is at the center of SBFT [22] which also features the usage of strong quorums.

Dashing3 tackles the new and subtle challenges due to weak certificates (which are not used in the above-mentioned protocols).

Multiple thresholds in a single timing model or two different timing models. Some Byzantine-resilient protocols such as UpRight [16, 24] 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 [30]) focus on the asynchronous-synchronous timing model [9, 10, 32, 35]. For instance, the recent work of Malkhi, Nayak, and Ren [32] and the work of Momose and Ren [35] 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 all different from these protocols. Our protocols are classic BFT protocols designed in a single timing model and assume the $f < n/3$ threshold for both safety and liveness. The different thresholds in our protocols are used to improve efficiency or robustness.

Parallel BFT using multiple BFT instances. Systems such as Mir-BFT [37] and the recent ISS [38] are beautiful and practical BFT systems aiming at running n parallel BFT instances for high throughput. By design, the two protocols are very different from Star: Mir-BFT and ISS work in the partially synchronous environment and rely on timers or Byzantine failure detectors, while Star is asynchronous if the underlying atomic broadcast component is asynchronous and uses a single atomic broadcast instance.

Handling parallel transactions using n BFT instances in one epoch turns out to be highly challenging. For instance, ISS can deliver transactions only when all BFT instances successfully terminate. In the full paper of ISS [39], ISS discusses how to select the n leaders for each epoch in the presence of failures and attacks to ensure liveness. In particular, ISS proposes three different and mutually exclusive policies for leader selection. These policies provide inherent trade-offs in terms of performance and robustness. Instead, Star has one BFT instance and does not have to deal with the issues.

3 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 *a-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 [19], 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 completely asynchronous environments if the underlying atomic broadcast protocol is asynchronous.

(Best-effort) broadcast. We use the term "broadcast" to represent that 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*, *tsign*, *tcombine*, *tverfiy*). *tgen* outputs a threshold signature public key and a vector of n private keys. A signature signing algorithm *tsign* 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 [11, 12]. One can also use a group of n signatures to build a (t, n) threshold signature for efficiency, as used in various libraries such as HotStuff [1, 41], Jolteon and Ditto [20], and Wendy [21]. The approach is also preferred for our protocols, as many of our protocols have more than one thresholds. (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 simply let $n = 3f + 1$

for this paper. 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.
Steps. We follow the standard definition of steps [13]. A step consists of receiving a message from some replica, running a local computation, and sending a message to some replica.

4 THE FAMILY OF DASHING PROTOCOLS

4.1 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 and 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 protocol must ensure 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 and broadcasts *commitQC*. Upon receiving the *precommitQC*, a replica becomes *locked* on b. Upon receiving the *commitQC*, a replica delivers b.

In case of view changes, each replica sends its latest *prepareQC* to the leader. Upon receiving a quorum of $n - f$ such messages, the leader selects the QC with the 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.

4.2 Overview of Dashing1

In Dashing1, we aim at using weak certificates (signatures from $f+1$ replicas) to improve on both efficiency and robustness. The reason

why weak certificates may improve efficiency is straightforward: replicas only need to collect $f + 1$ signatures and combine $f + 1$ signatures. It is less intuitive that one may use weak certificates to enhance BFT robustness.
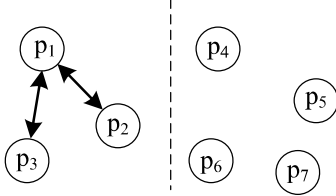

Figure 2: Dashing1 under network partition.

Fig. 2 describes a network partition scenario, where the leader p_1 can only receive messages from two other replicas (p_2 and p_3). As long as the partition persists, replicas in existing BFT protocols cannot make meaningful progress. For partially synchronous BFT protocols, replicas have to wait for their timers to expire and loop on view changes.

The idea of Dashing1 is that during network asynchrony, we allow replicas to proceed with weak certificates. When the network becomes synchronous and a transaction with a regular certificate is delivered, all transactions with weak certificates may be simultaneously delivered.

Note in the above approach, the leader may propose multiple blocks before the timer expires and collect multiple weak certificates. Even if a view change occurs, we still want to make the effort worthwhile. Namely, after the network becomes synchronous (either before or after the timer expires), transactions with weak certificates may be immediately delivered—all at a time.

Transforming the idea into a fully secure BFT protocol, however, is highly 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.

Second, we need to ensure that safety is still preserved in the presence of weak certificates. Namely, we should guarantee that if two conflicting blocks are extended from two conflicting branches, a regular certificate is formed for at most one of them. As shown in Fig. 3a, b_0 and b_0' are conflicting blocks and weak certificates 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 weak certificate 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 weak certificate (i.e., b_1'), a replica votes for block b_2' if and only if it has previously voted for the parent block b_1'. In this example, as $2f + 1$ replicas have already voted for b_1, it is impossible that $2f + 1$ replicas will vote for b_2'.

Third, we need to ensure that across view changes (or in the rotating leader mode), transactions with weak certificates can be processed. During view changes, we ask each replica to send its *highest* weak certificate to the new leader and the new leader can select a weak certificate to create a new proposal. After the proposed block is committed, all the blocks on the branch led by the block will be committed. However, we cannot simply let the new leader

(a) The safety challenge within a view.

(b) The safety challenge across views.
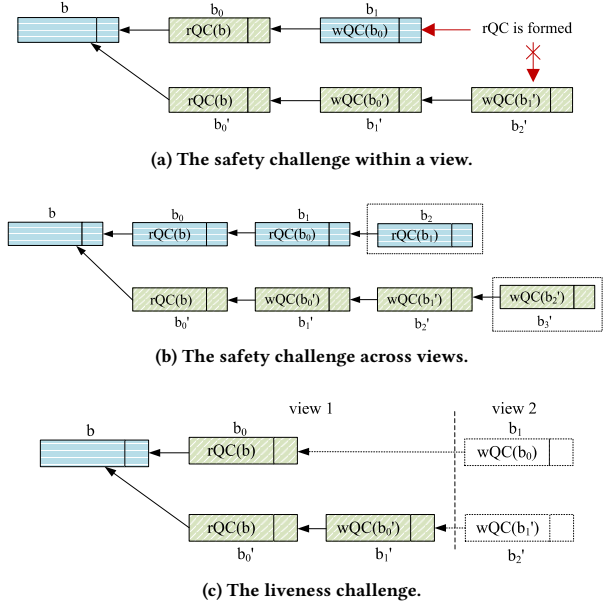

(c) The liveness challenge.

Figure 3: 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.)

select the highest wQC it receives due to a subtle safety problem. As shown in Fig. 3b, 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 a rQC for b_2 is also the *commitQC* for b_0. If a view change occurs and we let the leader select the highest weak certificate b_3', a proposal extending b_3' will be proposed. To attain liveness, all replicas need to vote for the proposal and b_3' will be committed. But b_0 has already been committed by at least one correct replica, violating safety. To address this issue, for any block b, we define *stable block* as the highest block for which a rQC has been formed on the branch led by b. Correspondingly, we require that each block b additionally maintains a *stable link* field sl which stores the hash digest of the stable block of b. (Note that the use of the stable link resembles the use of the parent link.) After the leader collects the certificates from the replicas, it will either select the highest rQC, or the wQC for which the stable block is the highest. 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. Upon receiving a proposal b', if b' extends a rQC for b, replicas decide whether to vote for b' by comparing b's stable block to its locked block (just as in HotStuff).

In fact, allowing the new leader to extend a weak certificate during view changes introduces a liveness challenge. Recall that in the normal case operation, we ask every replica to vote a block b that extends a weak certificate only if the replica has voted for the parent block of b. Unfortunately, we cannot enforce the same rule during view changes, as there may not even exist $f + 1$ correct replicas that have previously voted for the parent block of b. Fig. 3c illustrates an example where in view 1, the leader creates forks by creating multiple weak certificates, and in view 2, the new leader receives a weak certificate for b_0 or b_1' (or both). According to

the rule (for the normal case), the leader is allowed to extend a weak certificate and create a proposal (e.g., b_1 that extends b_0 or b_2' that extends b_1'). As b_0 and b_1' have been voted by only $f + 1$ replicas, there is no guarantee that either b_1 or b_2' will be voted by $f + 1$ correct replicas in view 1. In this scenario, a proposal from a correct replica will not be voted by any correct replica, creating a liveness issue. To address this challenge, we require a correct replica p_i decide whether to vote for a block extending a wQC (e.g., b_1) during view change by comparing the stable block of received block to the locked block of p_i. In the example, p_i voted for b_1 if p_i is not locked on a conflicting block of b (the stable block of b_1).

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 [1, 21, 36, 41].

4.3 Notation for Dashing Protocols

We specify the notation for the 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 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).

Messages. Messages transmitted among nodes are of the form $\langle type, block, justify \rangle$. We use three message types: GENERIC, VIEW-CHANGE, and NEW-VIEW. The VIEW-CHANGE and NEW-VIEW message are used during view change: VIEW-CHANGE messages are sent by replicas to the next leader, while NEW-VIEW message is sent by the new leader to the replicas. The $justify$ field of message m stores certificates to validate the $block$ contained in m. 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 QCBLOCK(qc) return the block b .

We have discussed two approaches to maintaining wQCs and rQCs (the last paragraph in Sec. 4.2). To hide the implementation detail, we let QCVOTE(m) denote the output of a partial signature signing algorithm for m or a conventional signing algorithm and let QCCREATE(M) be a QC generated from signatures in M. QCCREATE(M) may be a wQC or a rQC.

Rank of QCs and blocks. Following the notion in [20], we now define the $rank()$ function for QCs and blocks. $rank()$ does not return a concrete number. Instead, it takes as input two blocks/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 $cview$, the highest rQC QC_r, the highest wQC QC_w, the locked block lb, and the last voted block vb.

4.4 Dashing1

We present in Algorithm 2 and Algorithm 3 the normal case operation 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 weak certificates 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** $b.sl \leftarrow b'.sl, b.stable \leftarrow b'.stable$ **return** b
5   **if** qc is a rQC **then** $b.sl \leftarrow b.pl, b.stable \leftarrow b'$ **return** b
6 **procedure** STATEUPDATE(QC_w, QC_r, lb, qc)
7     $b' \leftarrow$ QCBLOCK(qc), $b'' \leftarrow b'.parent, b^* \leftarrow b''.parent, v \leftarrow b'.view$
8     $b_0 \leftarrow$ QCBLOCK(QC_w), $b_{high} \leftarrow$ QCBLOCK(QC_r)
9   **if** qc is a rQC
10   **if** $rank(b') > rank(b_{high})$ **then** $QC_r \leftarrow qc$
11   **if** $b'.stable = b''$ and $rank(b'') > rank(lb)$ **then** $lb \leftarrow b''$
12   **if** $b'.stable = b''$ and $b''.stable = b^*$ and $b''.view = b^*.view = v$
13   deliver the transactions in b^*
14   **if** qc is a wQC and $rank(b'.stable) \geq rank(b_0.stable)$ **then** $QC_w \leftarrow qc$

Algorithm 2: Normal case protocol for Dashing1 and Dashing2

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

Normal case protocol (Algorithm 2). We describe the chained version of the protocol. In each phase, the leader broadcasts a message to all replicas and waits for signed responses from replicas.

Algorithm 3: View change protocol for Dashing1 and 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    qc_high ← the rQC with the highest rank contained in M
5    b_1 ← QCBLOCK(qc_high), b ← CREATEBLOCK(b_1, cview, req, qc_high)
   ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
   │  for m ∈ M                                                     │
   │    if a wQC qc_d ∈ m.justify and QCBLOCK(qc_d) = d and        │
   │      rank(d.stable) > rank(b_0.stable) then vc ← qc_d, b_0 ← d │
   │    if rank(b_0.stable) ≥ rank(b_1) then b ← CREATEBLOCK(b_0, cview, req, vc), │
6  │      broadcast m = ⟨GENERIC, b, vc⟩                           │
   └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
7    else then broadcast m = ⟨GENERIC, b, qc_high⟩
8    //switch to normal case protocol
9  as a replica
10   wait for m = ⟨GENERIC, b, π⟩ from LEADER(cview)
11   b' ← b.parent, b_s ← b.stable, m ← ⟨GENERIC, b, ⊥⟩
12   if b'.view ≥ cview or rank(b') ≥ rank(b) or b.height ≠ b'.height + 1
13     discard the message
   ┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐
   │  if π is a wQC for b' and b_s = b'.stable and rank(b_s) ≥ rank(lb) │
14 │    vb ← b, STATEUPDATE(QC_w, QC_r, lb, π)                     │
   └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
15   if π is a rQC for b' and b_s = b' and rank(b_s) ≥ rank(lb)
16     vb ← b, STATEUPDATE(QC_w, QC_r, lb, π)
17   if vb = b then send QCVOTE(m) to LEADER(cview)
18   //switch to normal case protocol
19 ▷ NEW-VIEW phase: switch to NEW-VIEW phase if Δ_1 timeout occurs
```

At ln 9, the leader first proposes a new block b and broadcasts a $⟨\text{GENERIC}, b, qc_{high}⟩$ message, where qc_{high} is the last QC it receives (either a wQC or a 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). The timer is used to determine if the leader can form a rQC in time. The timer can be set as a small value or even zero in many circumstances and we will shortly discuss how to setup this timer. After Δ_2 expires, the leader combines the signatures in the votes into qc_{high} for the next phase.

Upon receiving a $⟨\text{GENERIC}, b, \pi⟩$ message from the leader, each replica p_i first verifies whether b is well-formed (ln 12-15), 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:

- If the π field is a wQC for b' (ln 16), 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 and creates a signature for b (Algorithm 1, ln 14).
- If π is a rQC for b' (ln 17-18), p_i verifies if b's parent block b' has a higher rank than vb. If so, p_i updates its local parameter QC_r to π and generates a signature. If b'' has a 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 a rQC for both b'' and b^* (the parent block of b''), then p_i commits block b^* and delivers transactions in b^* (Algorithm 1, ln 6-13).

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

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 $⟨\text{VIEW-CHANGE}, \bot, (QC_r, QC_w)⟩$ message to the leader (Algorithm 2, ln 23). Upon receiving $n - f$ VIEW-CHANGE messages, the leader first obtains a block b_1 with a QC that has the highest rank (ln 5). The leader then obtains a block 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 7).

Upon receiving a $⟨\text{GENERIC}, b, \pi⟩$ message from a new leader, each replica p_i verifies if the proposed block b extends a block of a prior view (ln 12-13). Then p_i votes for b if either of the following conditions is satisfied:

- b extends a block b' with a wQC (ln 14), 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 a rQC (ln 15-16), and the rank of the stable block of b is no less than that of the locked block of p_i.

On setting up the timers. In Dashing1, the leader maintains an additional timer Δ_2 to determine if it can receive enough signatures to form a rQC. One might wonder if the timer could impact system performance or system liveness. Indeed, if Δ_2 is too small, then it is possible that too many wQCs are formed and eventually a view change might be triggered. If Δ_2 is too large, the performance could be degraded accordingly.

For the first concern, we set up an upper bound on the number of wQCs that a leader can propose consecutively. A leader should form a rQC before sending to all replicas once the number of wQCs reach the bound.

For the second concern, we find that in the normal case (where there are no attacks), even if the timer Δ_2 is set as a very small value or even 0, the percentage that rQCs are formed among all QCs, perhaps surprisingly, is large. This is because while a replica is packing the proposal, other signatures may have been received by the replica and a rQC can be formed.

4.5 Dashing2

Dashing2 is a variant of Dashing1 and focuses on enhancing performance for the normal case. First of all, we use the idea of using $f + 1$ actively participating replicas for normal case operations (e.g., [26]). Instead of broadcasting a block to all replicas, the leader may choose to broadcast to only f replicas for wQCs. Note that the other $2f$ replicas now serve as only *backups*, thereby reducing the bandwidth consumption.

Periodically, the leader needs to switch to rQCs. In Dashing2, we set a parameter k: after k blocks with wQCs are proposed, the leader needs to collect a rQC for three phases so replicas can then deliver the k blocks with wQCs simultaneously. It is easy to find that Dashing2 eliminates the need of the Δ_2 timer, as the leader now collects wQCs for a fixed number of k phases.

Note in Dashing2, the leader should periodically switch between wQCs and rQCs. This is different from the strategy used in Dashing1 which sets an upper bound k on the number of consecutive wQCs: in Dashing1, the leader may simply collect a rQC before the upper

bound. The approach used in Dashing2 targets the normal case performance only and does not consider improving robustness under network partitions.
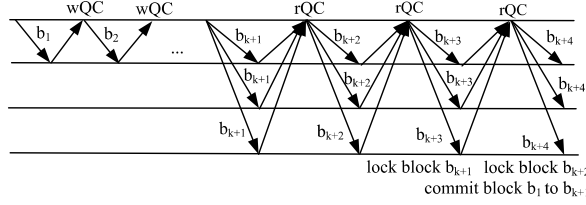

Figure 4: Periodically, Dashing2 collects wQCs for k blocks and then collects rQCs for one block.

4.6 Dashing3

We now show in Dashing3 how to further enable a fast path by using strong certificates (sQCs). Intuitively, supporting $3f + 1$ threshold may allow replicas to deliver the transactions in a single phase: if the leader collects a 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 [2, 3, 22], integrating sQCs in Dashing1, however, has its unique challenges due to usage of wQCs. Indeed, as a block supported by a sQC may be extended from a block with only a weak certificate, replicas cannot directly commit the block upon receiving a sQC. As depicted in Fig. 5, two conflicting blocks b and b' are proposed in the same view 1 with the same height. Moreover, a 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 a 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 on committed in view 3, in which case safety is violated as b'_2 is committed in view 2. As view change may occur at any moment, replicas cannot directly commit a block when a sQC is received.
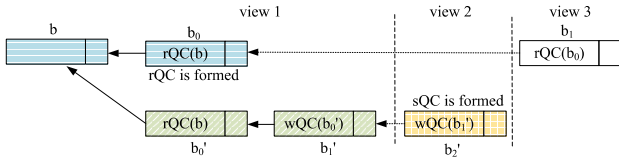

Figure 5: Challenge of integrating strong certificates in Dashing3.

In Dashing3, we treat a sQC for the first block proposed after view change as a 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 process now becomes similar to that in Fast-HotStuff [25] and Jolteon [20]. Accordingly, Dashing3 has $O(n^2)$ authenticator complexity and $O(n)$ message complexity. In addition, Dashing3 is a two-phase protocol with a one-phase fast path.

We show the pseudocode of Dashing3 in Appendix C. We make several major changes on top of Dashing1. First, if a replica p_i receives a strong certificate for block b from the leader, p_i directly commits b and delivers the transactions unless b is the first block proposed after the view change or b extends a block with a wQC. Second, 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 similar to the one used by the new leader to create the proposal. Replicas resume normal operations only after the NEW-VIEW message is verified. Third, for the first block b proposed after each view change such that a strong certificate is received, each replica treats it as a regular certificate and does not commit the block immediately. Finally, Dashing3 follows the two-phase commit rule that if a replica receives a regular certificate for both a block b and b' (the parent block of b), block b' can be committed. The concrete proof of correctness for Dashing3 is complex and we provide the proof in Appendix C.2.

4.7 Characteristics of the Dashing Protocols (Or: Benefits and Drawbacks for Individual Dashing Protocols)

We now summarize the characteristics of the Dashing protocols.

Dashing1. Dashing1 has two benefits compared to prior protocols: improved efficiency and enhanced robustness. First of all, we emphasize that being a partially synchronous BFT protocol, Dashing1, just as other such protocols, may not make progress in completely asynchronous environments. But Dashing1 offers an alternative way of handling some asynchrony scenarios.

Dashing1 excels in partition tolerance, because as long as there are $f + 1$ correct replicas in the primary partition then the system can proceed with wQCs. Similarly, Dashing1 offers benefits in intermittently asynchronous environments [34] or occasionally asynchronous environments. Due to these features, we view Dashing1 as a valuable addition to robust BFT protocols.

In one of our deployment scenarios, we set the view change timer as a value that is larger than the conventional ones in similar BFT protocols, in order to have better efficiency and robustness.

Dashing2. Dashing2 targets performance only and does not aim at improving robustness. It can no longer tolerate partitions or other asynchrony scenarios, but its design and implementation become neat: replicas just need to maintain regular timers for view changes. It is particularly well suited in scenarios where dedicated hardware and networks are used and failures are rare.

Dashing3. Dashing3 offers some interesting trade-offs among latency and throughput when compared to Dashing1. In general, Dashing3 has a more complex algorithm, a more subtle proof of correctness, and a more challenging implementation.

5 THE STAR FRAMEWORK

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

5.1 Overview of the Star Architecture

As in Narwhal and Tusk [17], the transmission and consensus processes in Star (as described in Fig. 6) are decoupled. The transmission process is fully parallizable 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. Replicas do not have to interact with the consensus process before advancing to the next epoch. The consensus process has only one BFT instance. The process 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.

The crucial difference between Star and the work [17] is that Star achieves less communication for both the gracious and uncivil scenarios. In particular, the transmission process in Star achieves $O(n^2)$ messages and $O(Ln^2 + \lambda n^2)$ communication, where L is the size of the proposal from each replica and λ is the security parameter. If being instantiated using HotStuff or PBFT, the consensus layer has $O(\lambda n)$ or $O(\lambda n^2)$ communication, respectively, and the total communication for Star remains $O(Ln^2 + \lambda n^2)$. Moreover, the linear communication pattern allows us to pipeline the message transmission process.

5.2 Star Details

The transmission process. The transmission process evolves in epochs. Each epoch consists of n parallel wCBC instances, as shown in Fig. 6a. 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, we can trivially solve the issue by asking each replica to deliver only the first wQC for each epoch.

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 that can validate the correctness of the proposal via the wQC. The above procedure is the "worst-case" scenario, where a correct replica stored a wQC but had no corresponding proposal. Even if the worst case 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 in 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 $\langle$PROPOSAL$, b, wqc\rangle$, 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 $\langle$PROPOSAL$, b_j, wqc_j\rangle$ 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 $W[e - 1]$.

Note we describe the code of the *obtain* function in the transmission process too, because only the transmission process has message queues. Jumping ahead, the *obtain* function takes as input wQCs a-delivered from the consensus process and outputs the corresponding proposals as delivered transactions.

Algorithm 4: The transmission process of Star (code shown for replica p_i; the chaining (pipelined) mode)

1 **initialization:** epoch number e, queue Q of pending transactions, received proposals *proposals*, the latest weak certificate wqc, and queue W of weak certificates are all initialized to $\perp$.
2 **func** *initepoch*(e)
3     $b.tx \leftarrow select(Q), b.epoch \leftarrow e$ //select a proposal b from Q
4   broadcast $\langle$PROPOSAL$, b, wqc\rangle$ //broadcast the proposal and wqc; pipelined mode
5   **upon** receiving a set M of $f + 1$ signed votes for b
6         $wqc \leftarrow$ QCCREATE(M)   //create a weak certificate
7   **wait until** $|proposals[e]| \geq n - f$ //enter the next epoch
8     $e \leftarrow e + 1$, *initepoch*(e)
9 **upon** receiving $\langle$PROPOSAL$, b_j, wqc_j\rangle$ from replica p_j in e for the first time
10   send signed vote for b_j to p_j
11     $proposals[e] \leftarrow proposals[e] \cup b_j$
12     $W[e-1] \leftarrow W[e-1] \cup wqc_j$ //certificates in the output queue
13 **func** *obtain*(e, m)   //obtain the union of proposals
14     $O \leftarrow \perp$ //to store delivered proposals
15   **for** wQC $qc \in m$
16   **if** QCPROPOSAL(qc) $\in$ *proposals*[e] //QCPROPOSAL(qc) returns the proposal for qc
17             $O \leftarrow O \cup$ QCPROPOSAL(qc)
18   **else** broadcast $\langle$FETCH$, e, qc\rangle$
19   **wait for** a PROPOSAL message containing QCPROPOSAL(qc)
20             $O \leftarrow O \cup$ QCPROPOSAL(qc)
21   clear $W[e]$, remove transactions in O from Q
22 **upon** receiving message $\langle$FETCH$, e, qc\rangle$ from replica p_j
23   **if** QCPROPOSAL(qc) $\in$ *proposals*[e]//fetch missing proposals; triggered by func *obtain*
24   send $\langle$PROPOSAL$,$ QCPROPOSAL(qc)$\rangle$ to p_j

---

Algorithm 5: The consensus process of Star

1 **initialization:** the epoch number of the current block le is initialized to 1. external: W //obtained from the transmission process
2 **upon** $|W[le]| \geq n - f$
3   *a-broadcast*($W[le]$)   //run the underlying atomic broadcast
4 **upon** *a-deliver(le, m)*
5     $O \leftarrow obtain(le, m)$
6   *deliver O*   //deliver the transactions in O in deterministic order
7     $le \leftarrow le + 1$

As there are at most n wCBC instances, the transmission process has $O(n^2)$ messages and $O(Ln^2 + \lambda n^2)$ communication.

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 in Algorithm 5, we use the *a-broadcast* and *a-deliver* primitives in atomic broadcast.
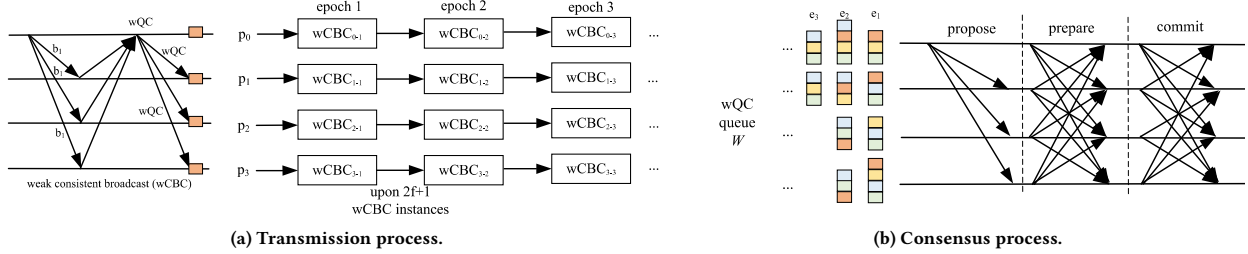
(a) Transmission process.

(b) Consensus process.

Figure 6: The Star architecture.

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 $W[le]$ 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 the transactions (from the transmission process) corresponding to wQCs a-delivered and delivers the transactions in deterministic order. If some proposals are missing, the replica may simply fetch the proposals from other replicas (via $obtain$). The fetch step is triggered only when needed and does not increase the message complexity or communication complexity of the whole protocol.

Censorship resilience (liveness) in Star. 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 specify the *select* function. HoneyBadgerBFT [34] invents a method that replicas randomly select transactions from their queue and use threshold encryption to achieve censorship resilience. EPIC [29] combines the conventional FIFO strategy used in [14] and the random selection strategy used in HoneyBadgerBFT to avoid threshold encryption. The completely asynchronous pattern in Star allows us to adopt the same approach in EPIC and achieve liveness under asynchrony. Note that doing so does not increase complexity.

In contrast, Mir-BFT and ISS devise interesting algorithms for liveness assuming partial synchrony. Note that an adversarial network scheduler can censor transactions for Mir-BFT and ISS.

Instantiating Star with PBFT (with good reason). When instantiating Star, we do not choose to use HotStuff or any Dashing protocol. This is because we cannot leverage the pipelining feature of these protocols with the Star framework and because using these protocols would incur higher latency. In contrast, PBFT allows the leader to propose new proposals without having to wait for the result from the last epoch, which makes it a better fit for Star. We comment using PBFT does not increase the message complexity or communication complexity for Star, because the transmission process in Star dominates the complexity for the protocol.

In Star, we use a variant of PBFT that is only 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. Second, during a view change, the new leader is not allowed to propose a nil block for any epoch number. Namely, for any epoch number e such that an agreement is not reached in a prior view, the new leader simply proposes $W[e]$. We describe the details of the protocol in Appendix D.

A Star variant. Star can be modified to support both wQC and rQC in the transmission process. The resulting protocol has a fast path for the consensus process: in the optimal case, we can reduce the number of phases from three to two (for both PBFT and HotStuff). While we did not implement the variant, we present the protocol variant in Appendix D.2.

6 IMPLEMENTATION AND EVALUATION

We implement all the four protocols introduced in this work and HotStuff in Golang using around 10,000 LOC, including 1,500 LOC for evaluation. We implement the chaining (pipelining) mode for the Dashing protocols and HotStuff. For Star, we implement a variant of PBFT for the consensus process as mentioned in Sec. 5.2. 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. We use digital signatures for quorum certificates, as in many prior works [1, 21, 36, 41].

We deploy the protocols in a local cluster with 40 servers (LAN) and also a popular cloud platform with 91 replicas where the servers are evenly distributed across five continents (WAN). In the LAN setting, each server has a 16-core 2.3GHz CPU and 128 GB RAM in the cluster. The network round-trip time between two servers is on average 5 ms. The network bandwidth is 200 Mbps. In the WAN setting, each instance has four virtual CPUs and 16 GB memory. The network round-trip time is 100-110 ms.

For each experiment, we use $3f + 1$ replicas and use f to denote the network size. We set the size for transactions and replies as 150 bytes. For all experiments for the Dashing protocols, we set the value of Δ_2 as 20 ms.

For readability, we first report the performance of Dashing1, Dashing3, and Star, and then compare Dashing2 (a variant of Dashing1) with Dashing1.

Performance (latency vs. throughput; throughput). We report the performance of Dashing1, Dashing3, Star, and HotStuff in both LAN (our local cluster) and WAN (cloud) settings.

(a) **Latency vs. throughput in LAN for** $f = 1$.

(b) **Latency vs. throughput in LAN for** $f = 10$.

(c) **Throughput in LAN for** $f = 1$.

(d) **Throughput in LAN for** $f = 10$.

(e) **Latency vs. throughput in WAN for** $f = 1$.

(f) **Latency vs. throughput in WAN for** $f = 10$.

(g) **Latency vs. throughput in WAN for** $f = 20$.

(h) **Latency vs. throughput in WAN for** $f = 30$.

(i) **Throughput in WAN for** $f = 1$.

(j) **Throughput in WAN for** $f = 10$.

(k) **Throughput in WAN for** $f = 20$.

(l) **Throughput in WAN for** $f = 30$.

(m) **Throughput of the protocols in WAN as** f **grows.**

(n) **Peak throughput under network delay for** $f = 1$.

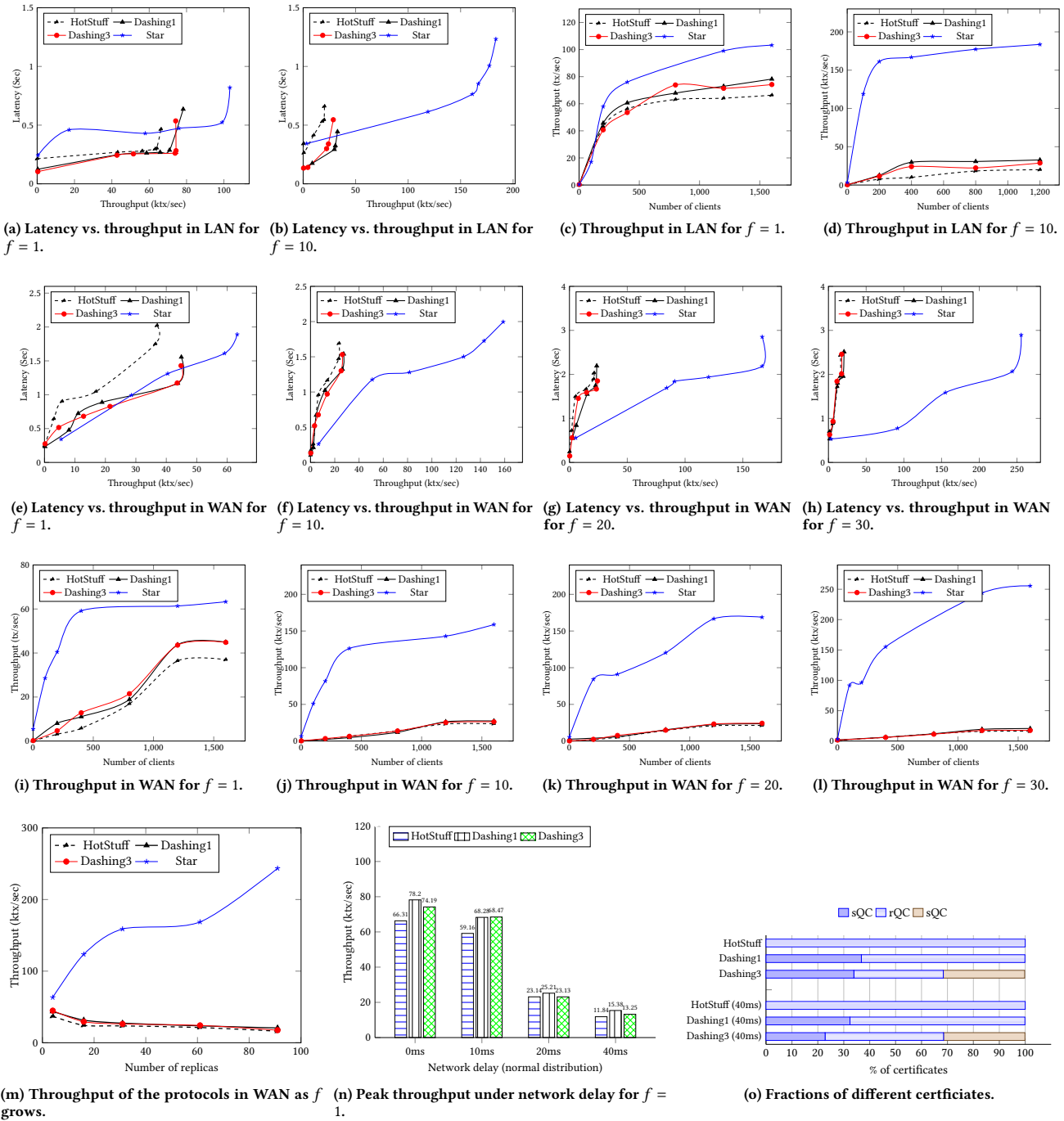(o) **Fractions of different certficiates.**

Figure 7: Performance of the protocols. Figures that compare only Dashing protocols and HotStuff are presented in Appendix A.

In the LAN setting, we report latency vs. throughput for $f = 1$ and $f = 10$ in Fig. 7a and Fig. 7b and throughput as the number of clients increases in Fig. 7c and Fig. 7d. Dashing1 and Dashing3 consistently outperform HotStuff. For $f = 1$, the peak throughput of Dashing1 and Dashing3 is 17.9% and 11.8% higher than that of Hot-Stuff, respectively. When $f = 10$, Dashing1 and Dashing3 achieve

63.8% and 43.7% higher peak throughput than HotStuff. In compari-son, Star significantly and consistently outperforms HotStuff and the Dashing protocols. For $f = 1$ and $f = 10$, the peak throughput of Star is 55.6% higher than HotStuff and 9x the throughput of HotStuff, respectively.

In the WAN setting, we report the performance of the proto-cols in Fig. 7e-7l. We also provide enlarged figures for Dashing1,

11

Dashing3, and HotStuff in Appendix A that do not contain Star. The performance of the protocols in the WAN environment is consistently lower than that in the LAN setting, due to the network delay. For the experiments we conduct in the WAN setting, all the Dashing protocols consistently outperform HotStuff, and Star more outperforms the Dashing protocols. In addition, the performance difference between Star and the family protocols in the WAN setting is much more significant than that in the LAN setting. In particular, when $f = 1$, the peak throughput of Dashing1, Dashing3, and Star are 21.5%, 21.2%, and 71.1% higher than that of HotStuff. When $f = 30$, the throughput of Dashing1 and Dashing3 are 28.1%, 8.8% higher than that of HotStuff; the throughput of Star is 15.8x the throughput of HotStuff.

While Dashing1 and Dashing3 provide some interesting performance trade-offs, they offer similar performance in most of the experiments. But Dashing3 has a fast path in the failure-free scenario, having lower latency in contention-free situations.

Scalability. We report in Fig. 7m the peak throughput of Dashing1, Dashing3, Star, and HotStuff in the WAN environment as f grows. All the Dashing protocols outperform HotStuff consistently. The peak throughput of Dashing1 is 29.9%, 16.2%, 10.7%, and 28.1% higher than that of HotStuff for $f = 5, 10, 20$, and 30, respectively. As discussed before, the performance improvement is due to the usage of wQCs.

For Dashing protocols and HotStuff, the throughput degrades as f grows, echoing the results for the HotStuff family of protocols. For instance, the peak throughput of Dashing1 degrades 52.7% for $f = 30$, compared to that for $f = 1$. In comparison, the peak throughput of Star actually grows as f grows. In particular, the performance of Star for $f = 30$ is 3.84x the throughput for $f = 1$. Meanwhile, the peak throughput of Star is 5.08x, 6.75x, 7.92x, and 15.8x the throughput of HotStuff for $f = 5, 10, 20$, and 30, respectively. 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 decoupled; 3) the transmission process is highly efficient and can be pipelined. When $f = 30$, the peak throughput of Star is 243 ktx/sec, in contrast to 16 ktx/sec for that of HotStuff and 20 ktx/sec for that of Dashing1.

Performance of the Dashing protocols under network delays. In the LAN setting, besides the original 5 ms delay, we inject delay of 10 ms, 20 ms, and 40 ms and random $+-10$ ms, $+-20$ ms, and $+-40$ ms normal distribution, respectively. For instance, for the 10 ms delay scenario, packets will experience a mean latency of $5 + 10$ ms with a standard deviation of 10 ms with a normal distribution.

We compare the performance of HotStuff, Dashing1, and Dashing3 and report the peak throughput (when the number of clients is 1200) for $f = 1$ in Fig. 7n. Compared to the performance in the LAN setting with no network delays, the performance of HotStuff degrades by 17.8%, 67,9%, 83.58% under 10ms, 20ms, and 40ms network delay, respectively. In contrast, the performance of Dashing1 degrades by 12.7%, 67.7%, and 80.3%. For Dashing3, the throughput degrades 7.7%, 68.8%, and 82.14%, respectively. Dashing1 and Dashing3 consistently outperform HotStuff, which highlights the robustness of our protocols under random network delays.

Additionally, we also report the fractions of wQCs, rQCs, and sQCs in the experiments with no delay and with 40ms delay in
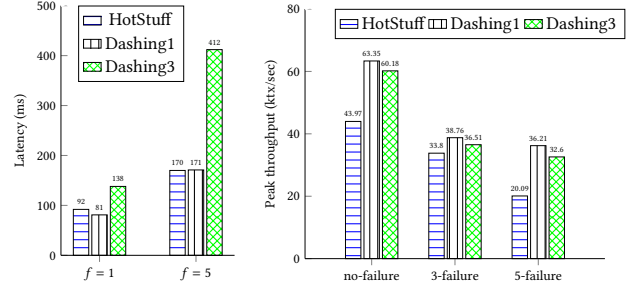
(a) View change latency. (b) Peak throughput of under failures ($f = 5$).

Figure 8: View change and performance under failures.
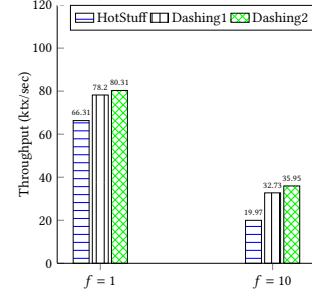

Figure 9: Dashing1 vs. Dashing2.

Fig. 7o. For Dashing1, the fraction of wQCs is 36.8% for the experiment with no delay and 32.4% for the experiment with 40ms delay. The finding explains why Dashing1 improves the performance of HotStuff. For Dashing3, its main benefit is reduced latency rather than improved throughput.

Performance under failures. We assess the performance under failures for Dashing1, Dashing3, and HotStuff. We evaluate the rotating leader mode, where the leader is rotated (view change) every second. We use 1200 clients in all the experiments.

We assess the average latency of view changes (due to failures) and report the results for $f = 1$ and $f = 5$ in Fig. 8a. In our experiments, the view change latency for Dashing3 is higher than Dashing1 and HotStuff, mostly because each NEW-VIEW message consists of $n - f$ VIEW-CHANGE messages, and replicas have to verify the messages accordingly. For $f = 5$, the view change latency of Dashing3 is about 2x the latency of Dashing1 and HotStuff.

We also report the peak throughput for $f = 5$ in Fig. 8b where there are no failures, three failures, and five failures, respectively. We fail the replicas at the beginning of each experiment for the experiments with failures. When there are failures, the performance for all three protocols degrades (due to view changes). When there are three failures, the performance of HotStuff, Dashing1, and Dashing3 is 23.1%, 38.8%, and 39.3% lower than the failure-free scenario, respectively. For the 5-failure scenario, the performance of HotStuff, Dashing1, and Dashing3 is 54.3%, 42.8%, and 45.8%. The performance of Dashing3 is consistently lower than Dashing1, largely because view change in Dashing3 is more expensive.

Dashing1 vs. Dashing2. We compare the performance of Dashing1 and Dashing2 in the LAN setting for $f = 1$ and $f = 10$. We set up the upper bound parameter k as 20 for Dashing2, where the leader broadcasts its proposal to only f replicas for k phases and then switch to collect rQCs. As depicted in Fig. 9, the performance

of Dashing2 is higher than Dashing1 due to the low network bandwidth consumption. For $f = 1$, the peak throughput of Dashing2 is 2.6% higher than Dashing1 and 21.1% higher than HotStuff. For $f = 10$, Dashing2 achieves 9.8% higher throughput than Dashing1 and 80.0% higher throughput than HotStuff.

7 CONCLUSION

We design and implement efficient BFT protocols using weak certificates, including a family of three Dashing protocols (that offer improved efficiency or robustness compared to HotStuff) and a new asynchronous BFT framework Star allowing proposing and processing parallel transactions using a single BFT instance. Via a deployment in both the LAN and WAN environments, we show that the our protocols outperform their counterparts. In particular, based on the test of the WAN setting with 91 replicas across five continents, the Star instantiation has a throughput of 243 ktx/sec, 15.8x the throughput of HotStuff.

REFERENCES

[1] 2022. HotStuff (Relab). https://github.com/relab/hotstuff. (2022).
[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting fast practical Byzantine fault tolerance. *arXiv preprint arXiv:1712.01367* (2017).
[3] Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. 2018. Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma. *arXiv preprint arXiv:1801.10022* (2018).
[4] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. 2021. Reaching Consensus for Asynchronous Distributed Key Generation. In *PODC*.
[5] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync hotstuff: Simple and practical synchronous state machine replication. In *S&P (Oakland)*. IEEE, 106–118.
[6] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the Symposium on Principles of Distributed Computing*. ACM, 337–346.
[7] Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. 2015. Making BFT protocols really adaptive. In *IPDPS*. IEEE, 904–913.
[8] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. 1994. Asynchronous secure computations with optimal resilience. In *Proceedings of the 13th annual symposium on Principles of distributed computing*. ACM, 183–192.
[9] Erica Blum, Jonathan Katz, and Julian Loss. 2019. Synchronous consensus with optimal asynchronous fallback guarantees. In *Theory of Cryptography Conference*. Springer, 131–150.
[10] Erica Blum, Chen-Da Liu-Zhang, and Julian Loss. 2020. Always have a backup plan: fully secure synchronous MPC with asynchronous fallback. In *Annual International Cryptology Conference*. Springer, 707–731.
[11] Alexandra Boldyreva. 2003. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In *PKC*. 31–46.
[12] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short signatures from the Weil pairing. *Journal of cryptology* 17, 4 (2004), 297–319.
[13] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2nd ed.).
[14] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*. Springer, 524–541.
[15] Christian Cachin and Marko Vukolić. 2017. Blockchain consensus protocols in the wild. In *DISC*. 1:1–1:16.
[16] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 277–290.
[17] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 34–50.
[18] Sisi Duan and Haibin Zhang. 2022. PACE: Fully Parallelizable BFT from Reproposable Byzantine Agreement. *IACR Cryptol. ePrint Arch.* (2022).
[19] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of ACM* 32, 2 (1988), 288–323.

[20] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2021. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. *arXiv preprint arXiv:2106.10362* (2021).
[21] Neil Giridharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. 2021. No-commit proofs: Defeating livelock in bft. *Cryptology ePrint Archive* (2021).
[22] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *DSN*. 568–580.
[23] Rachie Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The next 700 bft protocols. *ACM Transactions on Computer Systems* 32, 4 (2015), 12:1–12:45.
[24] Martin Hirt, Ard Kastrati, and Chen-Da Liu-Zhang. 2020. Multi-Threshold Asynchronous Reliable Broadcast and Consensus. In *24th International Conference on Principles of Distributed Systems*.
[25] Mohammad M. Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. 2021. Fast-HotStuff: A Fast and Resilient HotStuff Protocol. *arXiv preprint arXiv:2010.11454* (2021).
[26] Rüdiger Kapitza, Johannes Behl, Christian Cachine, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine fault tolerance. In *EuroSys*. 295–308.
[27] Ramakrishna Kolta, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva: speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems* 27, 4 (2009), 7:1–7:39.
[28] Klaus Kursawe and Victor Shoup. 2005. Optimistic Asynchronous Atomic Broadcast. In *ICALP*. 204–215.
[29] Chao Liu, Sisi Duan, and Haibin Zhang. 2020. EPIC: Efficient Asynchronous BFT with Adaptive Security. In *DSN*.
[30] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. 2016. XFT: Practical Fault Tolerance beyond Crashes. In *OSDI*. 485–500.
[31] Yuan Lu, Zhenliang Lu, and Qiang Tang. 2022. Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As Pipelined BFT. In *CCS*.
[32] Dahlia Malkhi, Kartik Nayak, and Ling Ren. 2019. Flexible byzantine fault tolerance. In *CCS*. 1041–1053.
[33] J.-P. Martin and L. Alvisi. 2006. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 202–215.
[34] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *Proceedings of the SIGSAC Conference on Computer and Communications Security*. ACM, 31–42.
[35] Atsuki Momose and Ling Ren. 2021. Multi-Threshold Byzantine Fault Tolerance. In *CCS*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1686–1699.
[36] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. 2021. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *SOSP*. 35–48.
[37] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-throughput BFT for blockchains. *arXiv preprint arXiv:1906.05552* (2019).
[38] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 17–33.
[39] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State-machine replication scalability made simple (extended version). *arXiv preprint arXiv:2203.05681* (2022).
[40] Gilad Stern and Ittai Abraham. 2020. Information Theoretic HotStuff. In *OPODIS*.
[41] Maofan Yin, Dahlia Malkhi, Micheal K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *PODC*.

A ADDITIONAL EVALUATION RESULTS

We report evaluation results for Dashing1, Dashing3, and HotStuff with enlarged figures in Fig. 10.

B PROOF OF CORRECTNESS

B.1 Correctness of Dashing1 and Dashing2

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 $\langle$GENERIC, $b, qc\rangle$, a correct replica votes for b only if (1) $b.stable = b'$ and qc is a rQC for b' (ln 17-19 of Algorithm 2 and ln 15 of Algorithm 3); or (2) $b.stable = b'.stable$ and qc is a wQC for b' (ln 16 of Algorithm 2
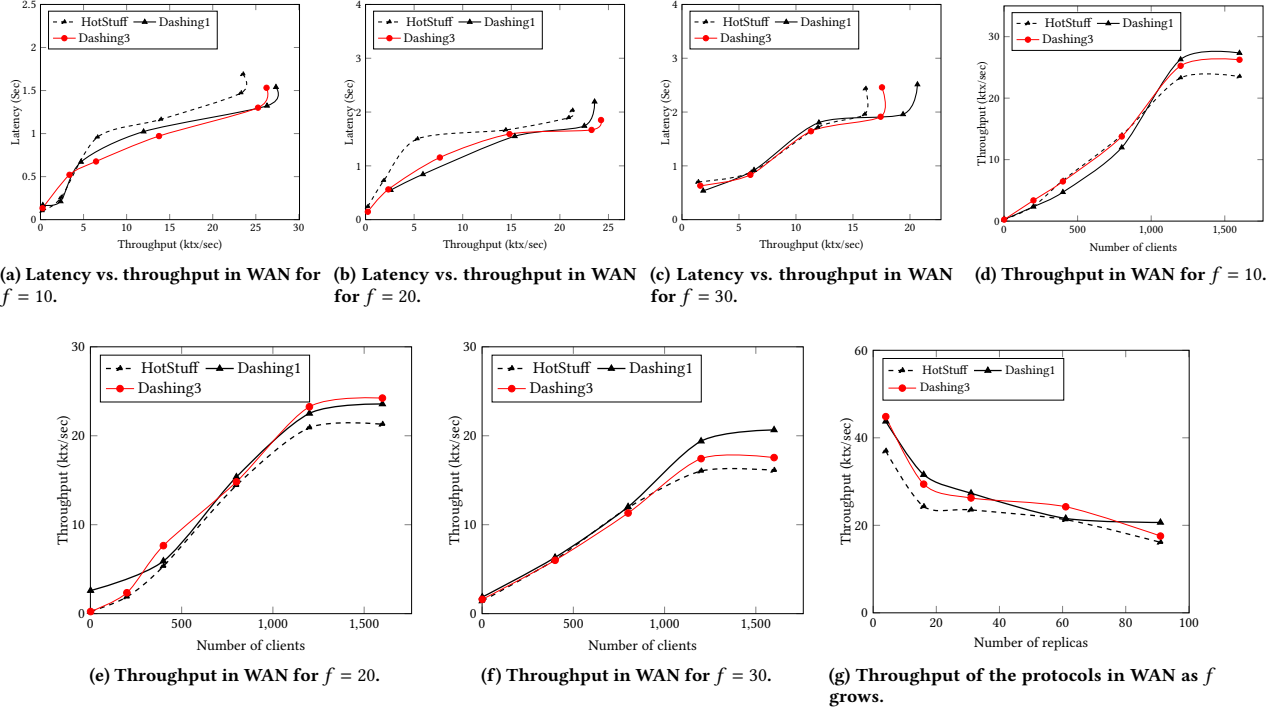
Figure 10: Evaluation results for Dashing1, Dashing3, and HotStuff with enlarged figures.

(a) **Latency vs. throughput in WAN for** $f = 10$.

(b) **Latency vs. throughput in WAN for** $f = 20$.

(c) **Latency vs. throughput in WAN for** $f = 30$.

(d) **Throughput in WAN for** $f = 10$.

(e) **Throughput in WAN for** $f = 20$.

(f) **Throughput in WAN for** $f = 30$.

(g) **Throughput of the protocols in WAN as** f **grows.**

and ln 14 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 a 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 B.1. *If b and d are two conflicting blocks and $rank(b) = rank(d)$, then a 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, a 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 a correct replica votes for at most one block with each height in the same view. □

LEMMA B.2. *Suppose that there exists a rQC or a wQC qc for b; 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 a 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 a 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 a 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 $d_c.parent = d$; a 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 $d_c.stable = d$. □

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

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 a 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. Therefore, at least one correct replica p_i has received a rQC qc for $d.stable$ before voting for d.

14

(2) qc is a wQC for d_0 and $d.stable = d_0.stable$. Let d_v denote the block with the 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, at least one correct replica p_i has voted for d_v' from Lemma B.2. Thus, we have $d_v'.stable = d_v.stable$ or $d_v'.stable = d_v$ according to Algorithm 2 (ln 16-18). Since $d_v.stable \neq d.stable$, $d_v'.stable \neq d_v.stable$. Then we know that $d_v'.stable = d.stabe = d_v$ and d extends $d.stable$. Meanwhile, p_i has received a rQC for d_v before voting for d_v'.

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

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

(1) The view of $b.parent$ is lower than v, where b be QCBLOCK(qc);
(2) If there exists a rQC for b_1 and $b_1.parent.view < v$, then b_1 equals b.

Proof. If a rQC is formed in view v, then there exists only one rQC qc with the lowest rank in view v (according to Lemma B.1).

(1) Let b denote QCBLOCK(qc) and b_v denote the block with the 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 B.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 a rQC for b_1, then at least a 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 $b_1 = b$. □

Lemma B.5. *If rQC qc for b is the rQC with the lowest height formed in view v and there exists a 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 with the 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 B.2, at least one correct replica has received a rQC for d_0. By Lemma B.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. □

Lemma B.6. *Suppose qc_1 and qc_2 are two rQCs, each is received by at least one correct replicas. 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, towards a contradiction, that $b_1.view = b_2.view = v$. According to Lemma B.5, we know that there exists a block b which is the block with the lowest height for which a 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 B.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 B.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, a 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 $b_1.view \neq b_2.view$. □

Lemma B.7. *If there exists a commitQC qc for b and a 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 denote $b.view$, v_d denote $d.view$, b'' denote QCBLOCK(qc), and b' denote $b''.parent$. 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 B.2, there exist rQCs for b, b', and b'' such that all these rQCs are received by at least one correct replica. Note that a 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 $|P| \geq f + 1$.

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 B.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 with the lowest height for which a 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 B.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 with 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 B.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 14 in Algorithm 3).
2) $b_0.stable = b_0'$, $m.justfy$ is a rQC for b_0', $b_0'.view < v_d$, and $rank(b_0') \geq rank(b_{lock})$ (ln 15 in Algorithm 3).

If condition 1) is satisfied, then according to Lemma B.3, b_0' is an extension of $b_0'.stable$ and at least one correct replica has received a rQC for $b_0'.stable$. Note that $rank(b_0'.stable) \geq rank(b_{lock})$. According to Lemma B.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 $m.justify$ is a rQC for b_0'. According to Lemma B.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 a rQC for d is formed in view v_d. According to Lemma B.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. □

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

Proof. Suppose that a $commitQC$ is formed for both b and d. According to Lemma B.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 B.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 B.7, a 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. □

Theorem B.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 with the highest rank locked by at least one correct replica. Note that a correct replica locks b_{high} only after receiving a $lockedQC$ qc for it. Let b_1 denote $\text{qcBlock}(qc)$. 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 a rQC for b_{high} and QC_r of the replica is the rQC with the 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 a 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 by all the correct replicas as conditions on ln 15 of Algorithm 3 are satisfied. In case (2), b will be voted by all the correct replicas as conditions on ln 14 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. □

C DASHING3

C.1 Dashing3 Details

Compared with Dashing1, a sQC is used as a certificate for a fast path in Dashing3. We present in Algorithm 7 and Algorithm 8 the normal case operation and view change protocol of Dashing3,

respectively. The utility functions are presented in Algorithm 6. Dashing3 follows the notation of Dashing1. rQCs and sQCs are collectively called *qualified* QCs in this section.

Normal case protocol (Algorithm 7). 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 replicas. qc_{high} is the last QC the leader receives (either a wQC, a rQC, or a sQC). After collecting $f + 1$ matching votes, the leader starts a timer Δ_2 (ln 7). The timer is used to determine if the leader can form a rQC or a 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', $b.height = b'.height + 1$, 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 18), 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 6, ln 13).
- If π is a rQC or a sQC for b' (ln 21-22), p_i verifies if the stable block of b is b', b' has a higher rank than vb, and b' has a higher 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 6, ln 10 and ln 15). If π is a 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 6, ln 11-12). If π is a sQC, 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 6, ln 14-15).

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

View change protocol (Algorithm 8). 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 7, ln 28). 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 6.

We now describe the procedure in more detail. Below, all number of lines is referred to as that in Algorithm 6. 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 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 5 of Algorithm 8).

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 $\text{SAFEBLOCK}()$, then p_i votes for b (ln 16 and ln 18).

Algorithm 6: Utilities for Dashing3

1 **procedure** $\text{CREATEBLOCK}(b', v, cmd, qc)$
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 $\bot$ **then** $b.sl \leftarrow b'.sl, b.stable \leftarrow b'.stable,$ **return** b
5   **if** qc is a rQC or a sQC **then** $b.sl \leftarrow hash(b')$ **return** b
6 **procedure** $\text{STATEUPDATE}(QC_w, QC_r, qc)$
7    $b' \leftarrow \text{QCBLOCK}(qc), b'' \leftarrow b'.parent,$
8    $b_0 \leftarrow \text{QCBLOCK}(QC_w), b_{high} \leftarrow \text{QCBLOCK}(QC_r)$
9   **if** qc is a rQC
10      $QC_r \leftarrow qc$
11   **if** $b'.stable = b''$ and $b''.view = b'.view$
12   deliver the transactions in b''
13   **if** qc is a wQC **then** $QC_w \leftarrow qc$
14   **if** qc is a sQC and $b'.stable = b''$ and $b'.view = b''.view$
15      $QC_r \leftarrow qc$, deliver the transactions in b'
16 **procedure** $\text{SAFEBLOCK}(M)$
17    $qc_{high} \leftarrow$ the qualified QC with the highest rank contained in M
18    $b_1 \leftarrow \text{QCBLOCK}(qc_{high}), b \leftarrow \text{CREATEBLOCK}(b_1, cview, req, qc_{high})$
19   **for** a wQC $qc \in M.justify$
20      $d \leftarrow \text{QCBLOCK}(qc), d' \leftarrow d.parent, d_s \leftarrow d.stable$
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 $rank(d) > rank(b_0)$
24          $vc \leftarrow qc, b_0 \leftarrow d$
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** $(b_2, \bot)$
28   **else if** $rank(b_0.stable) \geq rank(b_1)$ **return** (b_0, vc)
29   **return** (b_1, qc_{high})

C.2 Correctness of Dashing3

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 7, 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 a rQC or a sQC for b' (ln 21-23); or (2) $b.stable = b'.stable$ and qc is a wQC for b' (ln 18-20). In both cases, we say that qc and b are *matching*.

Let b' and b denote two consecutive blocks. In Algorithm 6, a replica p_i commits b only after receiving a certificate qc and one of the following condition is satisfied:

(1) qc is a rQC for b' such that $b'.stable = b'.parent = b$ and $b.view = b'.view$ (ln 9-12);

(2) qc is a 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 C.1. *Suppose a block b has been voted by a correct replica, then*

(1) any block d on the branch led by b has been voted 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)

Algorithm 7: Normal case protocol for Dashing3

1 **initialization:** $cview \leftarrow 1, vb, QC_w,$ and QC_r are initialized to $\bot$
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: $M \leftarrow \{\sigma | \sigma$ is a signature for $\langle \text{GENERIC}, b, \bot \rangle \}$
6   **upon** $|M| = f + 1$ **then** set a start timer Δ_2
7   **upon** Δ_2 *timeout* **then** $qc_{high} \leftarrow \text{QCCREATE}(M)$
8      $b \leftarrow \text{CREATEBLOCK}(b, cview, cmd, qc_{high})$
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 a rQC or a sQC **then** $QC_r \leftarrow qc_{high}$
12 **as** a replica
13   **wait for** $m = \langle \text{GENERIC}, b, \pi \rangle$ from $\text{LEADER}(cview)$
14    $b' \leftarrow b.parent, b'' \leftarrow b'.parent, b_s \leftarrow b.stable,$
15    $b_{gen} \leftarrow \text{QCBLOCK}(QC_r), m \leftarrow \langle \text{GENERIC}, b, \bot \rangle$
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 discard the message**
18   **if** π is a wQC for b' and $b.sl = b'.sl$ and $rank(b_s) \geq rank(b_{gen})$ and
19      $b_s.view = b''.view = b'.view = cview$ and $b' = vb$
20      $vb \leftarrow b, \text{STATEUPDATE}(QC_w, QC_r, \pi)$
21   **if** π is a rQC or a sQC for b' and $b.stable = b'$
22   and $rank(b') \geq rank(vb)$ and $rank(b') \geq rank(b_{gen})$
23      $vb \leftarrow b, \text{STATEUPDATE}(QC_w, QC_r, \pi)$
24   **if** $vb = b$ **then** send $\text{QCVOTE}(m)$ to $\text{LEADER}(cview)$
25 ▷ **NEW-VIEW phase:** switch to this line if Δ_1 *timeout* occurs
26 **as** a replica
27    $cview \leftarrow cview + 1$
28   send $\langle \text{VIEW-CHANGE}, vb, (QC_r, QC_w) \rangle$ to $\text{LEADER}(cview)$

Algorithm 8: View change protocol for Dashing3

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    $(b', qc) \leftarrow \text{SAFEBLOCK}(M), b \leftarrow \text{CREATEBLOCK}(b', cview, cmd, qc)$
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 $\text{LEADER}(cview)$
9    $b' \leftarrow b.parent, b_s \leftarrow b.stable, b_{gen} \leftarrow \text{QCBLOCK}(QC_r),$
10    $m \leftarrow \langle \text{GENERIC}, b, \bot \rangle$
11   **if** $b'.view \geq cview$ **or** $rank(b') \geq rank(b)$ or $b.height \neq b'.height + 1$
12   **discard the message**
13   **if** $M \in \pi$
14      $(b_p, qc) \leftarrow \text{SAFEBLOCK}(M), m \leftarrow \langle \text{GENERIC}, b, \bot \rangle$
15   **if** $b_p = b'$ and qc is a wQC or $\bot$ and $b.stable = b'.stable$
16   send $\text{QCVOTE}(m)$ to $\text{LEADER}(cview)$
17   **if** $b_p = b'$ and qc is a rQC or sQC and $b.stable = b'$
18   send $\text{QCVOTE}(m)$ to $\text{LEADER}(cview)$
19   //switch to normal case protocol
20 ▷ **NEW-VIEW phase:** switch to NEW-VIEW phase if Δ_1 *timeout* occurs

---

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 by at least one correct replica.

If $d \neq b$ and any block higher than d on the branch led by b has been voted by at least one correct replica, then we need to prove that d is voted 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 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 differentiate 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 7. 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 8, 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, a rQC or a 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, d has been voted 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 $d_c.view = d.view = v$, then d_c is proposed during normal case operation. According to ln 18 and ln 21 of Algorithm 7, 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 18-23 of Algorithm 7, if $d.parent.view < v$, then p_i votes for d_c only if p_i has received a rQC or a sQC for d and $d_c.stable = d$. □

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

PROOF. Assume, on the contrary, that $rank(b) = rank(d)$. 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 with each height during normal case operation, at least one of b and d is proposed during view change. Therefore, $b'.view < v$ or $d'.view < v$. Now we consider two cases:

(1) $b'.view < v$ and $d'.view < v$. According to Algorithm 8, 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) ($b'.view < v$ and $d'.view = v$) or ($b'.view = v$ and $d'.view < v$). If $b'.view < v$ and $d'.view = v$, then there exists a block d_0 with the lowest height on the branch led by d such that $d_0.view = v$. Hence, the view of $d_0.parent$ is lower than v. Let d'_0 denote a block on the branch led by d such that $d'_0.parent = d_0$. By Lemma C.1, at least one correct replica p_i has voted for d'_0. According to ln 18-23 in Algorithm 7, p_i has received a rQC or a sQC for d_0. Note that the view of $d_0.parent$ 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 situation is similar to the case where $b'.view = v$ and $d'.view < v$.

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, $rank(b) \neq rank(d)$. □

LEMMA C.3. *If a correct replica has voted for d and set its vb to d, then d must be an extension of $d.stable$ and at least one correct replica has received a qualified QC for $d.stable$.*

PROOF. Let d_0 denote $d.parent$. Let p_i denote a correct replica that has voted for d and set its vb to d. According to ln 16-23 of Algorithm 7, p_i has received a certificate qc for d_0, where qc and d are matching. We distinguish two cases.

(1) qc is a rQC or a sQC for d_0 and $d.stable = d_0$ (ln 21-23 in Algorithm 7). In this case, d is an extension of $d.stable$ and p_i received a qualified QC for $d.stable$.

(2) qc is a wQC for d_0 and $d.stable = d_0.stable$ (ln 18-20 in Algorithm 7). Let d_v denote the block with the lowest height on the branch led by d such that $d_v.stable = d.stable$. Let d'_v denote $d_v.parent$. Then $d_v.stable \neq d'_v.stable$. According to Lemma C.1, at least one correct replica p_j has voted for d_v. Since $d_v.stable \neq d'_v.stable$. Note p_j votes for d_v only if one of the following conditions holds: i) $d_v.stable = d'_v.stable$; ii) $d_v.stable = d'_v$ and p_i receives a qualified QC for d'_v. In this case, $d_v.stable = d.stable = d'_v$, d is an extension of $d.stable$, and p_j has received a qualified QC for $d.stable$.

Either way, d is an extension of $d.stable$ and at least one correct replica has received a qualified QC for $d.stable$. □

LEMMA C.4. *If a qualified QC is formed in view v, then there exists only one block b with the lowest rank for which a qualified QC is formed in view v, and we have that:*

(1) the view of $b.parent$ is lower than v;

(2) if there exists a qualified QC for b_1, $b_1.view = v$, and the view of $b_1.parent$ is lower than v, then b_1 equals b;

(3) if there exists a qualified QC for d and $d.view = v$, 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 with the lowest rank for which a qualified QC is formed in view v (according to Lemma C.2).

(1) Let b_v denote the block with the lowest height such that $b_v.view = v$ on the branch led by b. We have $b_v.height \leq b.height$ and the view of $b_v.parent$ is lower than v. If $b_v \neq b$, then there exists a block b'_v on the branch led by b such that $b'_v.parent = b_v$ and $b'_v.view = b_v.view = v$. From Lemma C.1, at least one correct replica p_i has received a rQC or a sQC for b_v. Thus, b_v is a block with a lower rank than b and a qualified QC for b_v is formed in view v, contradicting to 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 8, 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 with the 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 C.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. □

LEMMA C.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 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). □

LEMMA C.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 with the lowest height for which a qualified QC was formed in view v. Then according to Lemma C.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 C.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 C.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 with 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 to the condition that they are conflicting blocks. Therefore, $b_1.view \neq b_2.view$. □

LEMMA C.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 7, 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 C.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 C.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 C.1. Since correct replicas vote for at most one block with 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 C.6, $d.stable$ is equal to b or $d.stable$ is an extension of b. Hence, d is an extension of b. □

LEMMA C.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 a rQC for b', $b'.stable = b'.parent = b$ and $b'.view = v$; (2) qc is a 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 C.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 with the 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 by at least one correct replica p_i (Lemma C.1).

Let m denote the NEW-VIEW message for d_0. According to Algorithm 8, $m.justify$ 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 6 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 $\bot$.

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 24 in Algorithm 6).

ii) $d_0' = b_0$, i) is not satisfied and $rank(b_0.stable) \geq rank(b_1)$ (ln 25 in Algorithm 6).

iii) $d_0' = b_1$, i) and ii) are not satisfied (ln 26 in Algorithm 6) .

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, among which 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 C.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 a rQC. Let S denote the set of correct replicas who 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 C.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 a 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 with a higher rank, we have $rank(b_1) \geq rank(b.parent)$.

(a) If $rank(b_1) \geq rank(b)$, then from Lemma C.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 C.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 18-20 in Algorithm 7, 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 C.7. If $b_e'.view > v$, then the view of $b_e'.stable$ is higher than v. From Lemma C.3, b_e' is an extension of $b_e'.stable$ and a correct replica has received a qualified QC for $b_e'.stable$. According to 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. □

THEOREM C.9. (safety) If b and d are conflicting blocks, then they cannot be both committed, each by at least a correct replica.

PROOF. Suppose that there exist $commitQC$'s for both b and d. According to Lemma C.1, a qualified QC must have been formed for both b and d. From Lemma C.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 $rank(b) \neq rank(d)$, we assume w.l.o.g. that $rank(b) < rank(d)$. From Lemma B.7, we know that a qualified QC for d cannot be formed in view $d.view$. This completes the proof of the theorem. □

THEOREM C.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 $m.justify$ contains M, every correct replicas can verify the block b_v using function SAFEBLOCK() basing on input 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. □

D THE UNDERLYING BFT PROTOCOL IN STAR

D.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 $\langle$PRE-PREPARE, $v, B\rangle$ message to all replicas.

The block B is of the form $\langle v, cmd, height\rangle$, where v is the current view number, $B.cmd = W[le]$, and $B.height = le$. We directly use $B.height$ 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 $\langle$PRE-PREPARE, $v, B\rangle$ from the leader, a replica p_j first verifies whether 1) its current view is v, 2) $B.cmd$ consists of at least $n - f$ wQC or rQC for epoch e, and 3) p_j has not voted for a block $B.height$ in the current view. Then p_j broadcasts a signed PREPARE message $\langle$PREPARE, $v, hash(B)\rangle$. The replica also updates it W queue if any QC included in $B.cmd$ is not in $W[B.height]$.

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 $hash(B)$, replica p_j combines the messages into a regular certificate for B, called a *prepare certificate*. Then p_j broadcasts a $\langle$COMMIT, v, $hash(B)\rangle$ message. After receiving $n - f$ COMMIT messages with the same $hash(B)$, p_j *a-delivers B*.

Note that the PRE-PREPARE step and the COMMIT step carry only $hash(B)$ 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 $\langle$VIEW-CHANGE, C, $\mathcal{P}\rangle$, 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 $\langle$NEW-VIEW, $v+1$, c, M, $\mathcal{PP}\rangle$, 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 $\mathcal{PP}$ 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 $\langle$PRE-PREPARE, $v+1$, $h\rangle$, where h is the hash in the prepare certificate. If none of the replicas provides a prepare certificate, the new leader creates a $\langle$PRE-PREPARE, $v + 1$, $B\rangle$, 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 the normal operation.

D.2 A Star Variant

We now describe the variant of the protocol that has a fast path in the consensus protocol. The idea is to support both regular certificates and weak certificates in the transmission process.

In this variant, we modify the transmission process as follows. As in Algorithm 4, every replica additionally maintains a new local parameter rqc that is used to represent the latest rQC. We add the following procedures after ln 6: upon receiving $2f + 1$ matching votes, replica p_i creates a rQC and updates its rqc accordingly. At ln 4, a replica checks whether it has received a rQC for epoch $e - 1$. If so, it broadcasts a $\langle$PROPOSAL, b, $rqc\rangle$ message. Otherwise, it still broadcasts the $\langle$PROPOSAL, b, $wqc\rangle$ message.

Next, we modify the step 3 of the consensus protocol. If the proposed message B by the leader consists of n regular certificates, replicas can directly skip the commit step. Namely, after receiving $n - f$ matching PREPARE messages, replica p_j directly *a-delivers B*.

Now we describe the view change protocol. In the VIEW-CHANGE message, each replica additionally includes $\mathcal{L}$, a set of certificates for proposals. In $\mathcal{L}$, for any epoch number e between C and the replica's last vote, $W[e]$ is included. After receiving $n - f$ VIEW-CHANGE message, the leader additionally executes the following procedure. For each epoch number of any replica' last vote, if a pre-pare certificate is provided, the PRE-PREPARE message includes the corresponding block. If the $\mathcal{L}$ field in any VIEW-CHANGE messages consists of rQCs for proposals proposed in epoch e, the union of these rQCs will be packed in a block with a height e and broadcast in the PRE-PREPARE message. Otherwise, $W[e]$ is included.

E CORRECTNESS OF STAR

Basing 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 trans-actions in batches $\{$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 under-lying 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 E.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, the we have $V_i = V_j$.*

PROOF. Assume, towards a contradiction, that $V_i \neq V_j$. Let $\{qc_k\}_{k\in[1..n-f]}$ denote 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 $b_k = $ QCPROPOSAL(qc_k). Similarly, V_j is a union of transac-tions in proposals $\{b'_k\}_{i\in[1..n-f]}$, where $b'_k = $ 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). □

Now we are ready to prove safety.

THEOREM E.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_i denote the delivered transactions log of p_i. According to the safety of the consensus protocol, either L_i equals L_j or one of L_i and L_j is an an prefix of the other. Note that TL_i and TL_j contains transactions associated with messages in the a-delivered messages logs in a deterministic order. According to Lemma E.1, either TL_i

equals TL_j or one of TL_i and TL_j is an prefix of the other. This completes the proof of the theorem. □

THEOREM E.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 Honey-BadgerBFT), 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 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 the standard state machine replication mechanism such as state transfer to ensure all correct replicas deliver the transaction.) □

F CORRECTNESS OF THE STAR VARIANT

We prove the correctness of the Star variant as described in Appendix D.2. For safety, we prove that the consensus process is safe within a view and across views. For liveness, we prove that after GST, a correct primary is able to lead all the replicas to reach an agreement.

LEMMA F.1. *If B_1 and B_2 are different blocks that are proposed with the same epoch number in the same view and a prepare certificate is formed for both blocks, then $B_1 = B_2$.*

PROOF. We prove the lemma towards contradiction by assuming $B_1 \neq B_2$. Let v denote the view in which B_1 is proposed. As a valid prepare certificate consists of $2f + 1$ partial signatures, at least one correct replica has sent a PREPARE message for both B_1 and B_2 in view v. However, a correct replica votes for at most one block with a specific height in view v, a contradiction. □

LEMMA F.2. *If a one correct replica p_i has a-delivered block B_1 in view v with epoch number e, another correct replica has a-delivered a block B_2 in view v' with epoch number e such that $v' > v$, then $B_1 = B_2$.*

PROOF. If p_i has a-delivered B_1, it has received $2f + 1$ matching COMMIT messages (let the set of replicas be S_1), among which at least $f + 1$ are sent by correct replicas. Any of the $f + 1$ correct replicas have received a prepare certificate for B_1. As $v' > v$, we consider the NEW-VIEW message in view v'. As a valid NEW-VIEW message consists of $2f + 1$ VIEW-CHANGE messages (let the set of replicas be S_2), S_1 and S_2 has at least one correct replica p_i in common. According to the view change rules, replica p_i will include a prepare certificate for B_1 in its VIEW-CHANGE message. However, the leader in view v' has not received such a message so the leader proposes B_2, a contradiction.

Note that correctness holds even if a fast path occurs. During a fast path, a block B_1 with epoch number e consists of n rQCs. After receiving a prepare certificate for B_1, a correct replica p_i a-delivers B_1 directly. In this case, p_i knows that at least $f + 1$ correct replicas stores n rQCs for epoch number e. For any of the correct replicas, n rQCs will be included in VIEW-CHANGE messages in the $\mathcal{L}$ field. If in view v', a replica a-delivers B_2, a prepare certificate is formed. In other words, a correct replica has received n rQCs (for B_1) but has not sent it to the leader during the view change, a contradiction. □

As every block is a-delivered in order according to the epoch number of the delivered block, We prove safety for the consensus by proving the following theorem:

THEOREM F.3. *(safety) If a correct replica a-delivers a message m before a-delivering m', then no correct replica a-delivers a message m' without first a-delivering m.*

PROOF. Correctness in the same view follows from Lemma F.1 and correctness across views follows from Lemma F.2. That completes the proof. □

THEOREM F.4. *(liveness) If a correct replica a-broadcasts a message m, then all correct replicas eventually a-deliver m.*

PROOF. We consider two cases: a correct replica a-broadcasts a message m in the normal case protocol; a correct replica a-broadcasts a message m after view change. Correctness of the first case follows from the fact that all messages will be received after GST. We now show the correctness of the second case. After GST, a correct replica p_i is able to collect a set M of $n - f$ VIEW-CHANGE message for view v and broadcasts a NEW-VIEW message with a proposal m. Any PRE-PREPARE message included in the NEW-VIEW message includes either the hash of a block such that a prepare certificate is provided in the NEW-VIEW message, or $W[e]$, a set of wQCs. As prepare certificates and the wQCs/rQCs in $W[e]$ can be verified by any correct replica, then the proposal from p_i can be verified. Accordingly, any correct replica can then resume normal case operation and eventually a-deliver m. □