## HoтSтuff-1: Linear Consensus with One-Phase Speculation

Dakai Kang, Suyash Gupta<sup>‡</sup>, Dahlia Malkhi<sup>†</sup>, Mohammad Sadoghi University of California, Davis <sup>‡</sup>University of California, Berkeley <sup>‡</sup>University of Oregon <sup>†</sup>University of California, Santa Barbara

## **ABSTRACT**

This paper introduces HotStuff-1, a BFT consensus protocol that improves the latency of HotStuff-2 by two network-hops while maintaining linear communication complexity against faults. Additionally, HotStuff-1 incorporates an incentive-compatible leader rotation regime that motivates leaders to commit consensus decisions promptly.

HotStuff-1 achieves a reduction by two network hops by sending clients early finality confirmations *speculatively*, after one phase of the protocol. Unlike previous speculation regimes, the early finality confirmation path of HotStuff-1 is fault-tolerant and the latency improvement does not rely on optimism. An important consideration for speculation regimes in general, which is referred to as the *prefix speculation dilemma*, is exposed and resolved.

HotStuff-1 embodies an additional mechanism, *slotting*, that thwarts real-world delays caused by rationally-incentivized leaders. Leaders may also be inclined to sabotage each other's progress. The slotting mechanism allows leaders to drive multiple decisions, thus mitigating both threats, while dynamically adapting the number of allowed decisions per leader to network transmission delays.

## 1 INTRODUCTION

This paper introduces HotStuff-1, a Byzantine Fault-Tolerant (BFT) consensus protocol designed to reduce latency while simultaneously maintaining scalability, thus adhering to the most popular deployment of these protocols, blockchains [7, 61, 81]. Blockchains and decentralized systems employ a BFT consensus protocol because it enables them to provide their clients access to a verifiable, immutable ledger managed by multiple distrusting nodes, some of which may be malicious.

In this paper, we are interested in BFT consensus protocols for partially-synchronous setting, due to their safety against temporary network delays. Pioneering BFT consensus protocols belonging to the PBFT family [21, 35] employ a stable-leader design, where one replica designated as the leader is responsible for initiating a two-phase consensus algorithm that determines the ledger. There are several advantages to having a stable-leader design: the stable leader is incentivized to drive proposals promptly and can ensure low latency for clients. Furthermore, upon completing one phase, in which a quorum of votes is advertised, replicas can speculatively execute a proposal because the proposal is likely to become committed [31, 33, 41]. Unfortunately, the stable-leader design has some drawbacks. First, a dedicated leader increases censorship opportunities, as the leader decides what transactions to propose. Second, it also inhibits load and reward balancing among the replicas. Third,

when the leader fails, these protocols switch to a *view-change* algorithm that incurs quadratic communication to replace the leader (or change the view), causing a sharp drop in the system throughput [4, 23]. Last, the system throughput is bottlenecked by the available compute and network bandwidth at the leader.

Alternatives to the stable-leader design emerged in the blockchain world. First, Tendermint introduced a design that proactively replaces the leader at the end of each consensus decision [19]. Later, a framework known as HotStuff [83] reduced view-change communication costs to linear, and additionally *streamlined* protocol phases to (at least) double throughput. Thus, streamlined linear protocols in the HotStuff family mitigate the drop in system throughput by allowing regular leader replacement at (essentially) no communication cost. However, these protocols face three challenges:

- (1) Increased latency. Despite recent improvements (e.g., [58]) streamlined protocols incur higher latency than optimized, speculative-execution stable-leader protocols [31, 33, 41].
- (2) Leader-slowness phenomenon. In blockchain systems, regular leader replacement creates an undesirable incentive structure: rotating leaders may be inclined to delay proposing as close as possible to the end of their view expiration period in order to pick the transactions that offer the highest fees before proposing a block of transactions. Moreover, block-builders participating in a proposer-builder auction want to wait as long as possible to maximize MEV (maximal extractable value) exploits [24, 65, 67]. Thus, rational leaders/builders may slow down progress and cause clients to suffer increased latency.
- (3) Tail-forking attack. BeeGees [30] exposes another vulnerability of streamlined protocols, where faulty leaders can prevent proposals by honest leaders from being committed unless there are consecutive honest leaders. This attack surfaces when faulty leaders are interjected between honest leaders as leaders are rotated. While they may not succeed in completely censoring transactions, faulty leaders may cause specific clients to suffer increased latency and overall, slow down progress.

Thus, we are facing a conundrum: on the one hand, there are steady-leader protocols (PBFT-style) with quadratic communication, whose good-case latency is optimal, and they enable effective speculative execution. However, in addition to suffering quadratic communication in steady state, they are not *fair* and are vulnerable to censorship attacks. On the other hand, we have protocols (HotStuff-style) that proactively rotate leaders, have linear communication costs, and streamline protocol phases, but they are vulnerable to slowness and tail-forking attacks.

HotStuff-1 resolves these seeming trade-offs by introducing a BFT consensus solution that embodies two principal contributions:

- (1) A novel algorithmic core that is the first streamlined and linear BFT consensus algorithm that simultaneously combines speculative execution. HotStuff-1 acts as an optimist by speculatively executing client requests and serving the clients with the results of uncommitted transactions.
- (2) An adaptive slotting strategy that provides each leader with multiple slots to propose transactions. HotStuff-1 uses slotting to mitigate leader-slowness and tail-forking.

Early Finality Confirmation through Speculation. The algorithmic core of HotStuff-1 is a novel rotating-leader protocol that has early finality confirmation through speculation while maintaining linearity in the face of faults.

No prior approach has accomplished speculation with linear communication complexity with fault tolerance. Prior works belonging to the PBFT family, such as FaB [60], Zyzzyva [49], and SBFT [31], explore an *optimistically fast-path* approach to speculation. However, the optimistic fast-path of these protocols works only in failure-free runs and incurs quadratic communication upon failures. Protocols such as PoE [32, 33, 41] introduce speculation without relying on an optimistic fast-path, yet their design follows a stable-leader paradigm, and they incur quadratic communication upon a view-change.

HotStuff-1 achieves linearity with speculation by treating clients as first-class citizens of consensus, thereby allowing client *early finality confirmation*. Rather than forcing replicas to wait until they learn whether a transaction has committed, HotStuff-1 allows replicas to send commit-votes on transactions directly to clients precisely when a transaction can be committed by a quorum in HotStuff-2 [58]. Furthermore, when they vote to commit a decision, replicas *speculate* on the execution results and send responses to clients. Upon collecting responses from a quorum of replicas, clients can learn two things at once: a commit decision and its execution result, which enables an early finality confirmation.

Unsurprisingly, speculation notifications in HotStuff-1 come with a new challenge, a *speculation prefix dilemma*. The dilemma arises because clients, who receive commit-votes on a transaction lack context—internal to the consensus protocol—to learn from execution results of the transaction about preceding transactions that have become committed. Thus, we formulate a *Prefix Speculation rule* to guarantee safe speculation, which may be of benefit by itself for other speculation regimes.

In summary, HotStuff-1 achieves **early finality confirmation** with a **fault-tolerant fast-path** which incurs only **linear communication**.

Compared with previous approaches, HotStuff-1 strikes a middle-ground between the latency of HotStuff-2/SBFT and Zyzzyva, while maintaining linearity. More specifically, HotStuff-1 saves one (full) phase relative to the most efficient linear protocols, e.g., HotStuff-2, and a half-phase relative to SBFT, which is only optimistically linear. In HotStuff-2/SBFT, replicas must learn that a transaction has been committed before executing it and responding to clients. HotStuff-1 incurs an extra phase relative to Zyzzyva, because in Zyzzyva, replicas can respond to clients immediately upon a transaction being proposed. However, Zyzzyva incurs quadratic complexity against even a single failure. Hence, HotStuff-1 trades a slight sacrifice in latency for communication efficiency and fault tolerance.

HotStuff-1 comes in two regimes, **basic** and **streamlined** (itself a contribution over HotStuff-2 [58], which has a basic variant only), both of which are described in the body of the paper.

Low latency through slotting. Speculatively executing client transactions guarantees that in the good case, HotStuff-1 notifies clients early about transaction finality, but it does not resolve leader slowness and tail-forking attacks. We, therefore, go beyond the core HotStuff-1 algorithm and incorporate a novel slotting mechanism into HotStuff-1. Slotting allows each leader to propose multiple successive blocks of transactions; each leader has access to multiple slots and can propose one block of transactions per slot. Assigning more than one slot to a leader motivates a rational leader to ensure that its blocks commit quickly, opening the opportunity to propose more new blocks. Our slotting mechanism essentially brings the notion of a stable leader to the HotStuff family. However, having a fixed number of slots per leader still suffers from the slowness attack at the last slot of each leader. Thus, we devise an adaptive slotting mechanism, allowing a leader to propose as many slots as it can during a certain view period, and demoting leaders based on an adaptive timer mechanism.

We illustrate the practicality of our design by implementing two variants of HotStuff-1 (with and without slotting) in Apache ResilientDB (incubating) [9] and evaluating it against two baselines: HotStuff and HotStuff-2. Our results affirm that both the HotStuff-1 variants yield lower latency than the baselines; HotStuff-1 (without slotting) incurs up to 26.1% lower latency, while HotStuff-1 (with slotting) incurs 18.8% lower latency. Additionally, we illustrate the resistance of HotStuff-1 (with slotting) against leader-slowness and tail-forking attacks with up to 14.57× lower latency than the other protocols. In summary, we make the following contributions:

- (1) We introduce HotStuff-1, the first speculative streamlined BFT consensus protocol that treats clients as first-class citizens of the consensus process and serves them with early finality confirmations for their transactions.
- (2) HotStuff-1 uses speculation to send clients finality confirmations and execution results one phase earlier than HotStuff-2.
- (3) We expose and resolve a prefix speculation dilemma that exists in the context of BFT protocols that employ speculation.
- (4) We provide psuedocode for both the non-streamlined and streamlined variants of HotStuff-1.
- (5) We introduce slotting to HotStuff-1 to mitigate leader-slowness and tail-forking attacks faced by streamlined protocols.

#### 2 SYSTEM MODEL

We assume the system model adopted by existing partially synchronous BFT consensus protocols [21, 31, 33, 49, 84]. We assume a system of  $\bf n$  replicas, of which at most  $\bf f$  are faulty (malicious or crash-failed), and the remaining  $\bf n-\bf f$  replicas are correct;  $\bf n \geq 3f+1$ . Correct replicas follow the protocol: on the same input, produce the same output. This system receives requests from a set of clients; any number of clients can be faulty. We use R and C to denote a replica and a client, and each replica is assigned a unique identifier in the range  $\bf n$ 0 using function  $\bf id(R)$ .

**Authenticated communication**: each client/replica uses digital signatures to sign a message [45]. Additionally, replicas make use of BLS threshold signature scheme [18] to form (n, t) threshold

signatures. Each replica R has access to a private signature key, which it uses to create a signature share  $\delta_R$ . An aggregator needs only t shares out of  $\mathbf n$  to create the threshold signature. A receiver can use the corresponding public key to verify whether at least t replicas contributed to this signature. We use the notation  $\langle m \rangle_R$  to denote a signature or a threshold signature share on message m by replica R. Correct replicas only accept well-formed messages that have a valid signature. Further, we assume existence of collision-resistant hash function H(x), where it is impossible to find a value x', such that H(x) = H(x') [45].

**Adversary model**: Faulty replicas can delay, drop, and duplicate any message and collude with each other. However, a faulty replica cannot forge the identity/messages of a correct replica.

**Synchrony**: We assume a partial synchrony model [27] where there is a known bound  $\Delta$  on message transmission delays, such that after an unknown time called *GST* all transmissions arrive within  $\Delta$  bound to their destinations.

**System Guarantees**: The goal is for replicas to form an agreement on a global ledger of transactions requested by clients, and respond to clients with the outcome of executing transactions in sequence order. There are two requirements, *safety* is required under asynchrony and *liveness* requires synchrony/GST:

- (1) **Safety**: If two correct replicas, R and R' commit two transactions T and T' at sequence number k then T = T'.
- (2) Liveness: Each correct replica will eventually commit a transaction T.

## 3 EMBEDDING SPECULATION IN STREAMLINED PROTOCOLS

Our primary goal is to reduce the latency for partially-synchronous streamlined consensus protocols. That is, we aim to bring the latency of streamlined protocols to par with optimized stable-leader consensus protocols without losing a vital tenet: linearity. An additional goal of this work is to eliminate the slowness attack and mitigate the tail-forking attack from streamlined protocols.

To this extent, we design HotStuff-1, which uses two popular system design principles, speculation and slotting, to guarantee (1) low latency while maintaining linearity, (2) freedom from the slowness attack, and (3) in-frequent tail-forking attack. Thus, HotStuff-1 consists of two complementary components: (1) a speculative streamlined BFT consensus protocol, and (2) an adaptive slotting mechanism for streamlined consensus protocols.

In the rest of this section, we discuss HotStuff-1 and defer discussion on slotting till §6. To illustrate the challenges in introducing speculation to streamlined consensus protocols, we first briefly recap the skeleton of the HotStuff-2 [58] protocol.

## Recap of HotStuff-2.

HotStuff-2 optimizes HotStuff [83], the first streamlined BFT consensus protocol, by reducing commit latency by one phase (or two half-phases). Specifically, HotStuff-2 operates in a succession of *views* (Figure 1(i)). In each view, a leader proposes a transaction T and forms consecutive certificates on the initial proposal over two-and-half phases. In the first half-phase, the leader proposes the transaction T. In each subsequent phase:

- (1) Replicas generate threshold signature shares to ensure that at least  $\mathbf{n} \mathbf{f}$  replicas accept the leader's proposal and send it to the leader.
- (2) The leader aggregates threshold shares from n f replicas into a threshold signature, which we refer to as a *certificate*, and broadcasts it to all the replicas.

This chain of certificates guarantees safety as follows: The first certificate (prepare-certificate) guarantees non-equivocation by proving that it chains to a correct previous certificate and has the support of at least  $\mathbf{n}-\mathbf{f}$  replicas. The second is a commit-certificate, a certificate-of-certificate, guaranteeing that  $\mathbf{n}-\mathbf{f}$  replicas have locked the prepare-certificate, and despite any  $\mathbf{f}$  failures, T will be committed. Replicas that learn the commit-certificate can mark T committed, execute it, and return responses to the client; T becomes committed to the immutable ledger. These responses to the clients are often referred to as finality confirmations, as the corresponding transactions will never get revoked.

## Sending early finality confirmations.

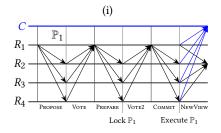
In the good case (no-failures), a HotStuff-2 client receives finality confirmations after two and half-phases (excluding the two network hops to receive client requests and to send a response to the client). With HotStuff-1, we want to cut down this delay to one and a half-phases. HotStuff-1 achieves this goal by making clients the first-class citizens of the consensus process—direct *learners* of consensus decisions and their execution results. HotStuff-1 requires replicas to employ speculative execution to serve clients with early finality confirmations.

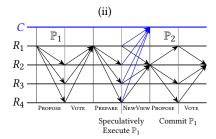
Rather than requiring replicas to wait till they learn whether a transaction *has committed*, HotStuff-1 allows replicas to speculate precisely when a transaction *can be committed* by a quorum in HotStuff-2. More specifically, replicas are allowed to speculate on a proposal in the second phase of the protocol, upon voting to commit a prepare-certificate. Replicas do not wait for a commit-certificate; they execute a transaction T as soon as they have the prepare-certificate for T and send a response to the clients. Thus, clients directly receive votes for commit and the result of executing T, which enables an early finality confirmation. When a client receives notifications from a quorum of  $\mathbf{n} - \mathbf{f}$  replicas, it learns two things: a transaction has been committed, and the execution result has been prepared in advance. Safety follows immediately from the commit-safety of HotStuff-2 because the client can determine whether a commit-certificate will form.

In a non-speculative protocol, a client needs to collect only f+1 execution responses/confirmations to determine the finality because replicas execute a transaction once the commit decision is reached. Consequently, in HotStuff-1, clients need to collect  $\mathbf{n}-\mathbf{f}$  responses because we are treating them as first-class citizens and serving them speculative responses, which do not guarantee that a commitment and finality will happen. A client learns that a transaction will finalize only upon collecting  $\mathbf{n}-\mathbf{f}$  responses. Figure 1 (ii) depicts our HotStuff-1 protocol.

## The Prefix Speculation Dilemma.

In HotStuff-1, when clients receive a quorum of responses for a transaction (say T), they learn that T will get committed and finality has been reached on adding T to the ledger in sequence order. The transactions preceding T in the sequence also become





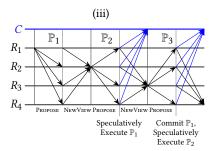


Figure 1: Workflows of (i) Basic HotStuff-2, (ii) Basic HotStuff-1, and (iii) Streamlined HotStuff-1

committed by this decision, and the result of executing T represents processing the *full prefix* of transactions up to and including T. However, the responses for T **must not** be combined with responses on transactions that precede T to form a quorum. That is, say T succeeds an earlier transaction T' in sequence order, T' < T. The commit votes of T' must not be combined with commit-votes of T' in forming a commit-decision on T'. (See Appendix §A.1 for a detailed explanation of why this breaks safety).

This brings forth a challenging dilemma with respect to speculation: the responses from T represent the execution of a full prefix ending with T. Hence, when a replica speculatively processes T, it must process all the transactions that precede it. But a replica must not send responses on preceding transaction T' to clients because clients are now first-class citizens and these responses represent commit-votes. If clients combine commit-votes from a partial quorum on T' with commit-votes from another partial quorum on T, they can mistakenly "learn" that a decision has been reached on T'. As noted above, this could break safety.

On the other hand, if the replica does not send results on T', then should T become committed, there would be a gap: the results from T were sent, but results from T' < T are missing.

Our policy for handling this dilemma, which is referred to as the Prefix Speculation Rule, is as follows:

Definition 3.1. **Prefix Speculation Rule.** A replica R can speculatively execute a transaction T if and only if T extends a prefix which is already known to commit.

Enforcing Prefix Speculation in HotStuff-1 is done as follows: when a replica receives a new certificate, it first checks whether any new commits can be applied. The replica proceeds to speculate on a potentially about-to-be committed block only if the transaction immediately preceding it is already committed.

#### Rollback.

Finally, we need to address the possibility that speculation does not succeed. Upon speculatively executing T, a replica R cannot commit T to the (global) ledger yet as it does not know if this transaction T will commit. Instead, each replica maintains a *local-ledger*, where it marks T prepared and executed. If in a succeeding view, R receives a prepare-certificate that does not extend T, then R must perform a *rollback* operation. This replica can observe that at least  $\mathbf{n} - \mathbf{f}$  replicas decided that T cannot commit and prepared another transaction T'. Specifically, the replica should now synchronize with the other replicas to fetch the transaction T', erase T from its

local-ledger, execute T', add an entry for T' to its local-ledger, and respond to the client. We discuss this in more detail in §4.2.

#### 4 SPECULATIVE CORE

We first describe the non-streamlined variant of HotStuff-1; in §5, we extend the description to the streamlined variant.

## 4.1 Non-Streamlined Speculation

As we treat clients as first-class citizens, we start by describing the client's behavior.

**Client Request.** When a client c wants the replicas to process its transaction T, it creates a Request message and broadcasts it to the replicas. This message includes the transaction T and the client's local timestamp (or a sequential request number). The client should ensure that timestamp values increase monotonically as replicas use these values to discard old or duplicate requests.

**Client Response.** When a client c receives identical Response messages from  $\mathbf{n} - \mathbf{f}$  replicas for its transaction T, it records this set of responses as an early finality confirmation for T, marks T as *executed* and accepts the result of execution.

**Replica psuedocode.** In Figures 2 and 3, we present the pseudocode for basic HotStuff-1. Prior to describing the algorithm in detail, we lay down some useful definitions.

Definition 4.1. Prepare-Certificate. A prepare-certificate  $\mathcal{P}(v)$  for a proposal m aggregates  $\mathbf{n} - \mathbf{f}$  threshold signature-shares for m in view v.

Definition 4.2. Commit-Certificate. A commit-certificate C(v) for a proposal m aggregates  $\mathbf{n} - \mathbf{f}$  threshold signature-shares for  $\mathcal{P}(v)$  in view v.

Definition 4.3. Highest Known Certificate. A certificate  $\mathcal{P}(v_{lp})$  for view  $v_{lp}$  is the highest prepare-certificate known to replica R if it does not have another certificate  $\mathcal{P}(w)$  for view w at R, such that  $w > v_{lp}$ . For brevity, we omit from the code explicitly updating  $v_{lp}$  every time R learns a new certificate.

Definition 4.4. Extending Certificates. Given two certificates  $\mathcal{P}(v)$  and  $\mathcal{P}(w)$ , for views v and w, at a replica R,  $\mathcal{P}(v)$  extends  $\mathcal{P}(w)$  if v>w and  $\mathcal{P}(v)$ 's construction includes hash of  $\mathcal{P}(w)$ . Further, if a certificate  $\mathcal{P}(k)$  extends  $\mathcal{P}(v)$  and  $\mathcal{P}(v)$  extends  $\mathcal{P}(w)$ , then transitively  $\mathcal{P}(k)$  extends  $\mathcal{P}(w)$ .

#### **Local state** (replica *R*):

- 1:  $\mathcal{P}(v_{lp})$ ,  $v_{lp}$ : always keeps the highest known prepare certificate and its view number
- 2:  $C(v_{lc})$ ,  $v_{lc}$ : always keeps the highest known commit certificate and its view number
- 3: v: current view
- 4: T: pending, uncommitted blocks of transactions
- 5: local-ledger, global-ledger

Figure 2: Local state on each replica of Basic HotStuff-1.

*Definition 4.5. Conflicting Certificates.* Given two certificates,  $\mathcal{P}(v)$  and  $\mathcal{P}(w)$ , for views v and w, at a replica R,  $\mathcal{P}(v)$  conflicts with  $\mathcal{P}(w)$  if neither  $\mathcal{P}(v)$  extends  $\mathcal{P}(w)$ , nor  $\mathcal{P}(w)$  extends  $\mathcal{P}(v)$ .

**Local state** at a replica includes: (1) highest prepare-certificate,  $\mathcal{P}(v_{lp})$ , formed in view  $v_{lp}$ , (2) highest commit-certificate,  $C(v_{lc})$ , formed in view  $v_{lc}$ , (3) current view v, (4) set of pending, uncommitted blocks of transactions  $\mathcal{T}$ , and (5) the local-ledger and the global-ledger.

**Propose.** When the leader  $\mathcal{L}_v$  for view v-a replica R with  $v = \operatorname{id}(R)$  mod  $\mathbf{n}$ -enters view v, it waits to receive NewView messages from at least  $\mathbf{n} - \mathbf{f}$  replicas. Each message carries the highest certificate known to its sender, which helps the leader learn the highest known certificate among them and update its  $v_{lp}$ . Additionally, if these  $\mathbf{n} - \mathbf{f}$  NewView messages contain threshold signature-shares for the highest  $\mathcal{P}(w)$ , the leader forms a commit-certificate C(w) (Figure 3, Line 7). Next, the leader aggregates client transactions (yet to be proposed) into a block  $B_v$  and creates a Propose message that includes the view number v,  $B_v$ ,  $\mathcal{P}(v_{lp})$ , and  $C(v_{lc})$  (if formed). Then,  $\mathcal{L}_v$  broadcasts this Propose message (lets call it m) to all the replicas (Lines 4-5).

**ProposeVote.** On receiving a Propose message m from the leader (Line 11), a replica R checks if one of the following holds:

- (1) The prepare certificate  $\mathcal{P}(w)$  in m matches the highest locked prepare certificate  $\mathcal{P}(v_{lp})$  for view  $v_{lp}$  at R, or
- (2)  $w > v_{lp}$ , in which case R updates its  $v_{lp}$  to w, sets  $\mathcal{P}(w)$  as highest known prepare certificate and runs the *recovery protocol* to fetch the block corresponding to  $\mathcal{P}(w)$  (§4.2).

If neither condition is satisfied, R ignores the message. Otherwise, R creates a ProposeVote message, which includes a threshold signature-share  $\delta_R^{\mathcal{P}}$  for m (includes hash of  $\mathcal{P}(v_{lp})$ ), and sends this message to the leader  $\mathcal{L}_v$  (Lines 11-15).

**Prepare.** When  $\mathcal{L}_v$  receives  $\mathbf{n}-\mathbf{f}$  well-formed ProposeVote messages for its proposal m, it tries to combine their signature shares into a threshold signature to create a prepare-certificate  $\mathcal{P}(v)$  for view v (Lines 8-9). If successful, the leader creates a Prepare message that includes the certificate  $\mathcal{P}(v)$  and broadcasts it to all the replicas (Line 10).

**Vote and Speculate on Prepare.** On receiving a PREPARE message from the leader, a replica R checks if the certificate  $\mathcal{P}(v)$  is valid, i.e., the threshold signature combines signature-shares from  $\mathbf{n} - \mathbf{f}$  replicas for the leader's proposal m. If it is valid, R updates its highest known prepare-certificate to  $\mathcal{P}(v)$ ; sets  $v_{lp} = v$ .

If  $B_v$ 's predecessor is already in the global-ledger (i.e., it meets the Prefix Speculation rule), R does the following (Lines 16-25):

(1) speculatively executes the transactions in block  $B_v$  of m.

```
1: event Upon PACEMAKER.ENTERVIEW(v) do
 2:
         Wait until received n - f NewView messages for view v
         Wait until v_{lp} == v - 1 or pacemaker. Share Timer(v)
        Let B_v be a block of client transaction yet to be proposed
 4:
        Broadcast m = \langle \text{Propose}, B_v, v, \mathcal{P}(v_{lp}), C(v_{lc}) \rangle_{\mathcal{L}_v}
 6: event Received \mathbf{n} - \mathbf{f} NewView messages with shares of C(w) do
        C(w) \leftarrow \text{CreateThresholdSign}(\mathbf{n} - \mathbf{f} \text{ distinct } \delta_R^C \text{ shares})
    event Received n - f ProposeVote messages do
        \mathcal{P}(v) \leftarrow \text{CreateThresholdSign}(\mathbf{n} - \mathbf{f} \text{ distinct } \delta_R^{\mathcal{P}} \text{ shares})
        Broadcast (Prepare, v, \mathcal{P}(v))_{\mathcal{L}_v}
     Backup role (running at each replica R (including leader)):
11: event Received (Propose, b_v, \mathcal{P}(w), \mathcal{C}(x))<sub>f_n</sub> do
        Execute all transactions up to (incl.) B_x, add result to global-ledger
        and respond to clients ⊳traditional-commit rule
13:
        if w \ge v_{lp} \triangleright vote \ to \ prepare \ B_v then
14:
            \delta_{R}^{\mathcal{P}} \leftarrow \texttt{CreateThresholdShare}(\mathcal{P}(w), Hash(B_v))
            Send (ProposeVote, v, \delta_R^{\mathcal{P}} \rangle_R to \mathcal{L}_v
16: event Received \langle \text{Prepare}, v, \mathcal{P}(v) \rangle_{\mathcal{L}_v} do
        if \mathcal{P}(v) extends \mathcal{P}(v-1) \triangleright prefix-commit rule then
18:
            Execute all transactions up to (incl.) B_{v-1}, add result to global-
            ledger and respond to clients
19:
        if predecessor of B_v is in global-ledger \triangleright Prefix Speculation rule then
20:
            if local-ledger state conflicts with B_v then
                Roll local-ledger back to the common ancestor
21:
22:
            Execute all transactions in B_v speculatively, add result to local-
            ledger and send client a response \triangleright speculatively execute B_v
        \delta_R^C \leftarrow \text{CreateThresholdShare}(\mathcal{P}(v_{lp})) \triangleright \textit{vote to commit } B_v
23:
        Send (NewView, v + 1, \mathcal{P}(v_{lp}), \delta_p^C)<sub>R</sub> to \mathcal{L}_{v+1}
        Call exitView()
26: event Upon timeout do
        Send (NewView, v + 1, \mathcal{P}(v_{lp}), \perp \rangle_R to \mathcal{L}_{v+1}.
        Call exitView()
29: function EXITVIEW() do
        v \leftarrow v + 1. \triangleright disable voting and speculative execution for view v
        Call PACEMAKER.COMPLETEDVIEW()
```

**Leader role** (running at leader  $\mathcal{L}_v$ ):

Figure 3: Basic HotStuff-1.

- (2) Replies to the respective clients.
- (3) Adds result of executing  $B_v$  to its local-ledger. In either case, R exits the view.

**ExitView and NewView.** A replica R is ready to exit view v in two cases: upon receiving a prepare message from the leader and upon a timer expiration. Prior to calling the EXITVIEW() function, R constructs a NewView message, which includes  $\mathcal{P}(v_{lp})$ , and forwards it to the leader  $\mathcal{L}$  for view v+1 (Line 30). It then invokes the pacemaker to orchestrate view-synchronization as needed.

**Commit.** There are two commit rules in basic HotStuff-1 (*traditional commit* and *prefix commit*), which dictate when a replica can write a block of transactions to the global-ledger.

Definition 4.6. Traditional Commit Rule. A replica marks a block  $B_{v-1}$  as committed, when it receives a commit-certificate C(v-1) for  $B_{v-1}$  with a proposal for higher view  $\geq v$ .

Definition 4.7. Prefix Commit Rule. A replica marks a block  $B_{v-1}$  as committed when in view v, it receives a prepare-certificate  $\mathcal{P}(v)$ , for proposal m by  $\mathcal{L}_v$ , that extends  $\mathcal{P}(v-1)$ .

As the name suggests, the traditional commit rule is common to any consensus protocol and has been used by all the protocols of the HotStuff family. Post speculatively executing the transaction, each replica calls the ExitView procedure where a replica creates a threshold share  $(\delta_R^C)$  on the prepare-certificate and forwards this threshold share with the NewView message to the leader of the next view (Lines 23-30). On receiving  $\mathbf{n}-\mathbf{f}$  threshold shares for the same prepare-certificate, the leader of the next view combines them into a commit-certificate and forwards it to all the replicas (Line 7). Upon receiving the commit-certificate C(v), a replica R adds the block  $B_v$  to the global-ledger and marks it committed (Line 12). Note: on receiving the commit-certificate, R replies to a client if R had not sent a speculative response for this transaction.

The prefix commit rule is an important optimization that allows correct replicas to commit blocks when HotStuff-1 is experiencing replica failures. We will expand on this in the next section.

## 4.2 Failures and Recovery Design

A malicious replica can impact the consensus in various ways if it is the leader for an ongoing view: (1) drop, delay, or prevent sending messages and/or certificates to prevent replicas from making progress, and (2) create two proposals that extend the same certificate to prevent replicas from having the same state. HotStuff-1 should quickly detect these failures and resolve them to prevent performance degradation.

## Detecting lack of progress: Timeouts

Like other protocols in the partial synchrony setting, HotStuff-1 requires replicas to set timers. A replica R starts a timer following the rules defined by the *pacemaker* protocol (§4.2.1). Upon timeout, a replica R assumes that the leader for the current view (say v) has failed and thus sends a NewView message to the leader for view v+1. Post this, R calls the ExitView procedure to move to the next view (Lines 26-28).

#### Lack of certificates of the last view

Leader  $\mathcal{L}_v$  of view v may fail to receive the prepare-certificate  $\mathcal{P}(v-1)$  due to an unreliable network or faulty behaviors of the the preceding leader. If it extends some other lower certificate (at a view (v-1)), its new proposal will get rejected by correct replicas that received and set  $\mathcal{P}(v-1)$  as the highest known prepare-certificate, and thus cannot form the prepare-certificate  $\mathcal{P}(v)$ . To ensure that the new proposal is accepted by all correct replicas,  $\mathcal{L}_v$  should wait for sufficiently long to receive the highest certificates known to all the correct replicas. Following the rules defined by the *pacemaker* protocol (§4.2.1), it is guaranteed that  $\mathcal{L}_v$  will receive the highest certificates after PACEMAKER.SHARETIMER(v) (Line 3).

#### Conflict Resolution: Rollback

When a replica R receives a prepare-certificate  $\mathcal{P}(v)$  for a message m, HotStuff-1 allows R to set  $\mathcal{P}(v)$  as the highest known certificate and speculatively execute transactions of block  $B_v$  in m. A faulty leader may not send  $\mathcal{P}(v)$  to other replicas, in which case  $B_v$  may not get committed. To ensure replicas have a common state (global-ledger), HotStuff-1 supports state rollback (or *erasing local-ledger*).

Definition 4.8. Rollback Condition. Given two transactions T and T', if a replica R speculatively executes T with prepare-certificate  $\mathcal{P}(w)$  in view w, R rolls back T if R receives a conflicting preparecertificate  $\mathcal{P}(v)$  for T' in view v, such that w < v.

Definition 4.8 tells a replica when it should rollback (or erase) its local-ledger. When a replica R receives a prepare-certificate  $\mathcal{P}(w)$  in view w for a proposal m, it speculatively executes m's transactions and only updates its local-ledger; R does not add m to the global-ledger as it has only received  $\mathcal{P}(w)$  for m and has no guarantee that m will commit in the future. Thus, R can erase its local-ledger and rollback the effects of m's transactions if it receives a certificate  $\mathcal{P}(v)$  for a conflicting proposal m' in view v, v > w (Lines 20-21).

#### **Prefix Commit: Processing Delayed Certificates**

Due to failures, replicas may vote on a proposal in a view but not receive a prepare-certificate for that proposal in the same view. For example, the leader for view v fails before broadcasting the prepare-certificate  $\mathcal{P}(v)$  for its proposal m to at least  $\mathbf{n} - \mathbf{f}$  replicas. If such is the case, neither the client will receive an early finality confirmation for m, nor the replicas will receive a commit-certificate for m in view v + 1. So, how can we decide the fate of m.

If m conflicts with another proposal m' proposed in a view w, w > v, then it will be rolled back as described earlier. However, if there are no conflicts, that is, the leader for some view x, x > v observes  $\mathcal{P}(v)$  and sets  $\mathcal{P}(v)$  as the highest known certificate for its proposal m', a replica R will execute transactions in  $B_v$  and reply to the client once R receives a commit-certificate C(x) for m' (Line 12). Fortunately, we have an *optimization* that allows replicas to execute and commit  $B_v$  at least one phase earlier; if x = v + 1, then a replica R can execute transactions in  $B_v$ , add them to the global-ledger, and reply to their clients (Line 17). We refer to this optimization as the prefix-commit rule.

#### **Recovery Mechanism**

A faulty leaders can skip broadcasting a certificate to all the replicas. If any future leader has access to this valid certificate, it can extend its new proposal from this certificate. Such scenarios can occur in any protocol of the HotStuff family and are not limited to just malicious attacks; for example, in HotStuff-2, a leader can crash fail before broadcasting the certificate to all the replicas.

If the leader  $\mathcal{L}_v$  for v extends its proposal m with the certificate  $\mathcal{P}(w)$  for view w, w < v and forwards this as a Propose message to all the replicas, then any replica R that receives this Propose message needs to validate  $\mathcal{P}(w)$  and requires access to the corresponding proposal (say m') proposed in view w. If R does not have access to m', then it can fetch it from  $\mathcal{L}_v$ , which  $\mathcal{L}_v$  should have because its proposal extends  $\mathcal{P}(w)$ .

 $<sup>^1</sup>$ See Appendix A.2 for a scenario illustrating this.

```
1: function ShareTimer(v) do
       return StartTime[v] + 3\Delta
3: function CompletedView() do
       if v \mod f + 1 \neq 0 then
4:
           Call EnterView(v)
5:
6:
           Call SynchronizeEpoch(v)
8: function SynchronizeEpoch(v) do
        \delta_R \leftarrow \mathsf{CreateThresholdShare}(v)
10:
       Send \langle \text{Wish}(v, \delta_R) \rangle_R to leaders \mathcal{L}_{v+k}, k = 0, 1, 2, ..., \mathbf{f}.
    Epoch Leader role (running at leader \mathcal{L}_{v+k}, k = 0, 1, 2, ..., f.):
11: event Upon receiving \mathbf{n} - \mathbf{f} Wish messages of view v do
        TC_v \leftarrow \text{CreateThresholdSignature}(\mathbf{n} - \mathbf{f} \text{ distinct } \delta_r \text{ shares})
12:
       Broadcast TC_v.
13:
    Epoch Backup role (running at each replica R):
14: event Upon receiving TC_v at time t do
       Relay TC_v to the leaders \mathcal{L}_{v+k}, k = 0, 1, 2, ..., \mathbf{f}
15:
        for k \leftarrow 0, 1, 2, ..., f do
16:
           StartTime[v+k] \leftarrow t+k\tau
17:
       Call EnterView(v)
```

Figure 4: Pseudocode of Pacemaker Protocol

4.2.1 **Pacemaker.** For a system to make *progress*, at least  $\mathbf{n} - \mathbf{f}$  correct replicas should be in the same view. Otherwise, a leader cannot collect enough votes to make progress and to generate a prepare-certificate (§5). Specifically, under an unreliable network or when the leader is faulty, correct replicas can diverge: some replicas may have progressed to higher views, while others are stuck on an old view. To prevent this divergence among correct replicas, we adopt the *pacemaker* designs of prior works [22, 51]; group views into *epochs*, each of which contains  $\mathbf{f} + 1$  consecutive views.

In Figure 4, we illustrate the pseudocode for pacemaker. Every time a replica R reaches at the end of a view, it calls the function Completed (Lines 3-7) to check if the next view (say v) is part of the current epoch. If this is the case, R enters view v. Otherwise, v is the first view of the next epoch (v mod f+1=0) and R must synchronize its view with the other replicas. R calls the function SynchronizeView(v) (Lines 8-10) and delays entering the view v until the view synchronization is complete.

The function SynchronizeView(v) requires R to send a Wish message to the f+1 leaders of the next epoch;  $\mathcal{L}_{v+k}$ , where k=0,1,2,...,f. When a leader for the next epoch receives  $\mathbf{n}-\mathbf{f}$  Wish messages for view v, it creates a *Timeout Certificate TC*<sub>v</sub> and broadcasts it to all the replicas (Lines 14-15). Any non-leader replica R that receives  $TC_v$  forwards this certificate to all the f+1 leaders for the next epoch. Next, R sets the *starting time* for each of the next f+1 views v+k, k=0,1,2,...,f. Say, R received  $TC_v$  at time t, then view v+k starts at time  $t+k\tau$ , where  $\tau$  is a predetermined timer length that is sufficiently long for a non-faulty leader to reach a consensus on the proposal of its view. *Note:* the starting time for view v+k is also the timeout for view v+k-1. Post this, R enters the next view v (Lines 16-18).

Such a pacemaker guarantees that, after GST, once the first synchronization is done at view  $v_s$ , if a correct replica sets it timer for view  $v,v \geq v_s$  to expire at time  $t+\tau$ , then all correct replicas will enter view v before  $t+2\Delta$  and no correct replica will time out and enter view v+1 before  $t+\tau-2\Delta$ , where  $\Delta$  is the transmission delay bound [22, 51]. If the leader  $L_{v'}$  for view v' waits for an additional message delay,  $\Delta$ , after  $StartTime[v']+2\Delta$ , then it is guaranteed to receive NewView messages from all the correct replicas and learn the highest known certificate. Thus, the function ShareTimer(v) returns after  $StartTime[v']+3\Delta$ .

#### 5 STREAMLINED SPECULATION

Basic HotStuff-1 (§4.1) processes only one proposed batch of transactions every two phases. Like HotStuff, we can *streamline the phases* of HotStuff-1 to ensure that we rotate leaders and inject a new batch of transactions in each phase. This has the potential to increase throughput by 2×.

Borrowing from the streamlined variant of HotStuff, streamlined HotStuff-1 works as follows: it overlaps the second phase of view v, consisting of *Prepare* and *NewView* steps, with the first phase of view v+1, namely, *Propose* and *ProposeVote* steps. Each view (or leader) lasts for only one phase. The leader for each view waits for  $\mathbf{n} - \mathbf{f}$  NewView messages from the preceding view. The leader first attempts to create a prepare-certificate from the threshold shares it received from the replicas. It then selects the highest preparecertificate it knows and references it in a new proposal carrying a new batch of client transactions.

**Commit Rule.** Unlike the basic HotStuff-1, the streamlined design has only one commit rule: replicas follow the prefix commit rule (Definition 4.7) to add a transaction to the global-ledger. As each view consists of one phase, there is no explicit opportunity to create a commit-certificate. In view v, a replica R commits a block  $B_{w-1}$ , proposed in view w-1, if the proposal for view v includes the certificate  $\mathcal{P}(w)$  that extends the certificate  $\mathcal{P}(w-1)$ . Note: We no longer distinguish between prepare and commit certificates as in basic HotStuff-1.

**Speculation.** Replicas may speculate on a block B when it is about-to-be committed according to the prefix commit rule. That is, a replica R can speculate on a block B upon receiving a proposal carrying  $\mathcal{P}(B)$  if  $\mathcal{P}(B)$  is from the immediately preceding view; hence, the replica is preparing to commit B.

**Prefix Speculation Execution.** As in the basic variant, Prefix Speculation is needed in streamlined HotStuff-1 to resolve the Speculation Prefix dilemma explained above. Appendix A.3 illustrates the dilemma for the streamlined variant. The enforcement of the Prefix Speculation rule is similar to the basic regime: a replica R can speculate on a block B provided that P(B) extends a certificate P(B'), where B' is committed.

### 5.1 Streamlined HotStuff-1 Protocol

The streamlined protocol is reduced into a single phase of (1) *propose* and (2) *vote* that includes the speculative execution as demonstrated in Figure 1(iii).

**Propose.** When the leader  $\mathcal{L}_v$  for view v-the replica R with  $v = \mathrm{id}(R) \mod n$ -receives well-formed NewView messages from

```
Leader role (running at leader \mathcal{L}_v):
 1: event Upon Pacemaker.EnterView() do
2:
        Wait until received n - f NewView messages for view v
        Wait until v_{lp} == v - 1 or ShareTimer(v)
3:
        Let B_v be a block of client transaction yet to be proposed
4:
        Broadcast m = \langle \text{Propose}, B_v, v, \mathcal{P}(v_{lp}) \rangle_{\mathcal{L}_v}
 6: event Received \mathbf{n} - \mathbf{f} NewView messages with shares of \mathcal{P}(w) do
        \mathcal{P}(w) \leftarrow \text{CreateThresholdSign}(\mathbf{n} - \mathbf{f} \text{ distinct } \delta_R \text{ shares})
     Backup role (running at each replica R (including leader)):
 8: event Received (Propose, B_v, \mathcal{P}(w))_{\mathcal{L}_n} do
        if \mathcal{P}(w) extends \mathcal{P}(w-1) >commit-rule then
9:
           Execute all transactions up to (incl.) B_{w-1}, add result to global-
10:
           ledger and respond to clients
        if w == v - 1 and predecessor of \mathcal{P}(v - 1) is in global-ledger \triangleright Prefix
11:
        Speculation rule then
           if local-ledger state conflicts with B_{v-1} then
12:
               Rollback local-ledger to the common ancestor
13:
14:
           Execute all transactions in B_{v-1} speculatively, add result to local-
           ledger and send client a response \triangleright speculatively execute B_{v-1}
        if w \ge v_{lp} then
15:
           \delta_R \leftarrow \texttt{CreateThresholdShare}(\mathcal{P}(\textit{vlp}), \textit{Hash}(\textit{B}_\textit{v}))
16:
           Send (NewView, v + 1, \mathcal{P}(v_{lp}), \delta_R \rangle_R to \mathcal{L}_{v+1}
17:
        Call exitView()
18:
19: event Upon timeout do
        Send (NewView, v + 1, \mathcal{P}(v_{lp}), \perp)<sub>R</sub> to \mathcal{L}_{v+1}.
20:
        Call exitView()
21:
22: function EXITVIEW() do
        v \leftarrow v + 1. \triangleright disable voting for view v
23:
        Call pacemaker.completedView()
24:
```

Figure 5: Streamlined HotStuff-1.

at least  $\mathbf{n}-\mathbf{f}$  replicas, it tries to combine their threshold signature-shares into a threshold signature to create a certificate  $\mathcal{P}(w)$  for view w, where w < v. If  $\mathcal{L}_v$  is successful, it updates its highest known certificate  $\mathcal{P}(v_{lp})$ . Otherwise,  $\mathcal{P}(v_{lp})$  is updated with the highest certificate carried by the  $\mathbf{n}-\mathbf{f}$  NewView Messages. In either case, the leader extends its highest certificate to form its new proposal as a Propose message and broadcasts it to all the replicas. This proposal includes the view number v, a block  $B_v$  of client transactions yet to be proposed, and  $\mathcal{P}(v_{lp})$ .

**Execute and Ledger Update.** On receiving a Propose message (lets call it m) from the leader (Line 8), a replica R checks if the view w, for the certificate  $\mathcal{P}(w)$  in m, is less than  $v_{lp}$ , in which case R ignores the message. Otherwise, R sets  $\mathcal{P}(w)$  as the highest known certificate  $\mathcal{P}(v_{lp})$ , and if R had speculatively executed transaction at  $v_{lp}$ , then it rollbacks its state.

Next, *R* does the following (Lines 9-18):

(1) Following the commit-rule: if  $\mathcal{P}(w)$  extends  $\mathcal{P}(w-1)$ , then R executes transactions for all blocks up to  $B_{w-1}$  (blocks that  $B_{w-1}$  extends) if yet to be executed, adds them to the global-ledger and sends a reply to respective clients.

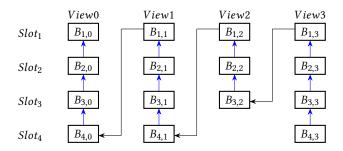


Figure 6: Adaptive Slotting

- (2) Following the Prefix Speculation Rule: if the block extended by  $B_w$  has been committed, then R speculatively executes the transactions in blocks  $B_w$ , adds them to the local-ledger and sends a reply to respective clients.
- (3) Finally, R creates a NewView message, which includes a threshold signature-share  $\delta_R$  for m (includes hash of  $\mathcal{P}(v_{lp})$ ), and sends this message to the leader of the next view,  $\mathcal{L}_{v+1}$ .

**Timer expiration.** In case of timer expiration, replica R constructs a NewView message, which includes an empty threshold signature-share and the highest known certificate  $\mathcal{P}(v_{lp})$ , and forwards it to the leader  $\mathcal{L}$  for view v+1.

**ExitView and NewView.** Like earlier, a replica R is ready to exit view v in two cases: upon receiving a Propose message from the leader and upon a timer expiration. ExitView() invokes the pacemaker to orchestrate view-synchronization as needed.

#### 6 SLOTTING

Rotating leaders in BFT protocols leads to the following challenges:

- (1) **Leader-slowness phenomenon.** Leaders may be inclined to delay proposing as close as possible to the end of their rotation because they are incentivized to include transactions that yield higher fees and thus maximize their MEVs.
- (2) **Tail-forking attack.** In streamlined protocols, the two protocol phases are necessary to commit a transaction spread across the reign of two leaders. The second leader, if malicious, may skip the proposal by the previous leader by pretending it did not receive enough votes for it instead of helping drive it to a commit decision.

We resolve these two challenges by adding *slotting* to the core of streamlined consensus protocols. Slotting provides each leader with opportunities to propose multiple blocks, one per slot, until their rotation time. Each leader incorporates an adaptive slotting mechanism to propose as many slots as possible within the allotted view timer.<sup>2</sup>

Assigning multiple slots to each leader/view: (1) motivates the leader to pack any available transactions and not wait for transactions to arrive in the future, and (2) eliminates opportunities for tail-forking attacks for all but the last slot in each view.

<sup>&</sup>lt;sup>2</sup>Since the number of slots may not be fixed, each correct leader may further tag its last slot as the final slot to gracefully invoke the pacemaker of correct replicas to advance their views.

**Local state** (replica *R*):

- 1:  $\mathcal{P}(s_{lp}, v_{lp}), s_{lp}, v_{lp}$ : always keeps the highest known prepare certificate and its slot number and view number
- 2: s: the slot in the current view
- 3:  $\delta_h$ : always keeps the highest signature-share sent

Figure 7: Additional Local State in Streamlined-HotStuff-1 with Slotting.

## 6.1 Slotting Design

We proceed to describe how to incorporate a slotting design into streamlined HotStuff-1. *Note*: our design of slotting is applicable to any protocol of the HotStuff family.

We introduce two additional notations:

First, we enumerate leader proposals with a pair of numbers: a leader/view number and a slot number within the view. Blocks are ordered lexicographically: if v < v', then block  $B_{i,v}$  is ordered before  $B_{i',v'}$ . If v = v' and i < i', then block  $B_{i,v}$  is ordered before  $B_{i',v'}$ . For instance, in Figure 6, we illustrate a chain of blocks generated under the slotting design. Each block extends a certificate of the preceding one, resulting in a snake-like chain that threads blocks within each view and, at the end of each view, threaded to the next view. In the Figure, block  $B_{2,1}$  includes a certificate for  $B_{1,1}$ ,  $B_{1,2}$  of  $B_{4,1}$ , and so on. Second, we introduce a new message type, NewSlot, to distinguish between when a new leader starts a new view and when the current leader starts a new slot.

Next, we describe the modifications needed to support slotting. **Local state** at a replica includes slot state (refer to Figure 7), in addition to view state: (1) highest known certificate,  $\mathcal{P}(s_{lp}, v_{lp})$ , formed in view  $v_{lp}$ , slot  $s_{lp}$  (2) current view v and current slot s (3) the highest signature share  $\delta_h$  it sent. As before, a replica maintains pending, uncommitted blocks of transactions, a local-ledger and the global-ledger.

Figure 8 illustrates the pseudocode of streamlined HotStuff-1 with slotting.

**Propose.** At each slot, the leader  $\mathcal{L}_v$  for view v awaits messages from at least  $\mathbf{n} - \mathbf{f}$  replicas of either of the following types:

- (1) well-formed NewView messages for view v 1, if i = 1, or
- (2) well-formed NewSlot messages for slot (i-1,v) if i>1. Thus,  $\mathcal{L}_v$  administers two types of transitions.

*NewView*: The first is entering a new view. The leader awaits well-formed NewView messages for view v-1 from at least  $\mathbf{n}-\mathbf{f}$  replicas. If all messages are non-empty and contain threshold signature-shares of the same slot  $s^*, s^* \geq 1$ , of some previous view  $v^*$ , the leader can combine these shares into a threshold signature to create a certificate  $\mathcal{P}(s^*, v^*)$ .

However, like in earlier sections, a replica R's timer for view v-1 may expire before it receives the last slot from the leader of v-1. In such a case,  $\mathcal{L}_v$  fails to receive  $\mathbf{n}-\mathbf{f}$  NewView messages for the same slot and view and cannot form a certificate using the NewView messages. In Section 6.2, we explain the details of what  $\mathcal{L}_v$  does if it fails to form a certificate.

Then,  $\mathcal{L}_v$  proposes slot (1, v) and extends the highest certificate  $\mathcal{P}(s_{lp}, v_{lp})$  known to it.

NewSlot: For slot (i, v), where i > 1, the leader  $\mathcal{L}_v$  awaits well-formed NewSlot messages from at least  $\mathbf{n} - \mathbf{f}$  replicas for slot

```
Leader role (running at leader \mathcal{L}_v):
 1: event Upon PACEMAKER.ENTERVIEW() do
         Keep updating (s_{lp}, v_{lp}) until (1) received \mathbf{n} - \mathbf{f} NewView messages
         for view v each with signature-share \delta not greater than (s_{lp}, v_{lp}) or
         (2) n = 3f + 1 NewView messages for view v messages are received
         or (3) ShareTimer(v)
        Let B_{1,v} be a block of client transactions yet to be proposed
         Broadcast m = \langle \text{Propose}, B_{1,v}, 1, v, \mathcal{P}(s_{lp}, v_{lp}) \rangle_{\mathcal{L}_v}
 5: event Received \mathbf{n} - \mathbf{f} NewView messages with shares of \mathcal{P}(s^*, v^*) do
         \mathcal{P}(s^*, v^*) \leftarrow \text{CreateThresholdSign}(\mathbf{n} - \mathbf{f} \text{ distinct } \delta_R \text{ shares})
     event Received \mathbf{n} - \mathbf{f} NewSlot messages with shares of \mathcal{P}(s, v) do
         \mathcal{P}(s, v) \leftarrow \text{CreateThresholdSign}(\mathbf{n} - \mathbf{f} \text{ distinct } \delta_R \text{ shares})
 9:
         Let B_{s+1,v} be a block of client transaction yet to be proposed
         Broadcast m = \langle \text{Propose}, B_{s+1,v}, s+1, v, \mathcal{P}(s,v) \rangle_{\mathcal{L}_v}
     Backup role (running at each replica R (including leader)):
11: event Received (Propose, B_{s,v}, \mathcal{P}(s_w, w)\rangle_{\mathcal{L}_n} do
        if \mathcal{P}(s_w, w) extends \mathcal{P}(s_w - 1, w) \triangleright commit-rule-case1 then
             Execute all transactions up to (incl.) B_{s_w-1,w}, add result to global-
             ledger and respond to clients
         else if s_w = 1 and \mathcal{P}(s_w, w) extends \mathcal{P}(s^*, w - 1) \triangleright commit-rule
         case2 then
15:
             Execute all transactions up to (incl.) B_{s^*,w-1}, add result to global-
             ledger and respond to clients
         if (s = s_w + 1 \text{ and } v = w) \text{ or } (s = 1 \text{ and } v = w + 1)
         and predecessor of \mathcal{P}(s_w, w) is in global-ledger \triangleright Prefix Speculation
         rule then
17:
             if local-ledger state conflicts with B_{S_{W},W} then
                Rollback local-ledger to the common ancestor
18:
             Execute all transactions in B_{s_w,w} speculatively, add the result to
19:
             local-ledger and send the client a response > speculatively execute
        if ((s = 1) \text{ or } (s_w = s - 1 \text{ and } w = v \triangleright Consecutive-Slot-Rule))
20:
              and (w > v_{lp} \text{ or } (w = v_{lp} \text{ and } s_w \ge s_{lp})) then
             \delta_R \leftarrow \text{CreateThresholdShare}(\mathcal{P}(s_{lp}, v_{lp}), Hash(B_{s,v}))
21:
22:
             Send (NewSlot, s, v, \mathcal{P}(s_{lp}, v_{lp}), \delta_p \rangle_R to \mathcal{L}_v
         s \leftarrow s + 1 \triangleright disable voting for slot s
23:
24: event Upon timeout do
         \text{Send } \\ \overline{\langle \text{NewView}, v+1, \mathcal{P}(s_{lp}, v_{lp}), \delta_h \rangle_R} \text{ to } \mathcal{L}_{v+1}.
25
         v \leftarrow v + 1, s \leftarrow 1. \triangleright disable voting for view v
        Call PACEMAKER.COMPLETEDVIEW()
```

Figure 8: Streamlined HotStuff-1 + Slotting.

(i-1,v). Once it collects enough votes from the previous slot, it combines the signature-shares of the replicas into a threshold signature to create a certificate  $\mathcal{P}(i-1,v)$ . *Note:* slots do not expire; hence, these  $\mathbf{n} - \mathbf{f}$  messages always form a certificate. The leader proceeds to propose slot (i,v).

**ProposeVote.** A replica R follows an almost identical logic to streamlined HotStuff-1 in processing a leader proposal (say m) and voting for it. If s = 1, R assumes that m is a proposal for the first slot

of view v; R checks if the certificate  $\mathcal{P}(s_w, w)$  is lexicographically ordered greater than or equal to  $\mathcal{P}(s_{lp}, v_{lp})$ . Otherwise, if s > 1, then R follows the Consecutive-Slot Rule, which prohibits a leader to skip slots or proposals in its view.

Definition 6.1. **Consecutive-Slot Rule.** In a given view v, a replica R will vote for the leader's proposal at slot s, extending certificate  $\mathcal{P}(s_w, v)$ , if  $s_w = s - 1$ .

**Slot-change.** There is no timer for individual slots within a view: given a view v, a replica exits slot s upon receiving a well-formed leader proposal for slot (s, v), which extends  $\mathcal{P}(s-1, v)$ .

**View-change.** A lack of progress is detected at the view level (not at the slot level). When the timer for view v-1 expires, a replica R exits view v-1; R uses the pacemaker to synchronize entering to view v and sends a NewView message containing the  $\delta_h$ , the highest signature-share it sent.

**Commit Rule.** The same as the streamlined design without *slotting*, Streamlined HotStuff-1 with *slotting* has only one commit rule: replicas follow the prefix commit rule (Definition 4.7) to add a transaction to the global-ledger.

However, as we form a two-dimension chain with *slotting*, there are two different cases when a replica R learns a new certificate  $\mathcal{P}(s_w, w)$  and commits the block extended by  $\mathcal{P}(s_w, w)$ : (1)  $s_w > 1$ : commits block  $B_{s_w-1,w}$  if  $\mathcal{P}(s_w, w)$  extends  $\mathcal{P}(s_w-1, w)$ . (Line 12) (2)  $s_w = 1$ : commits block  $B_{s^*,v^*}$  if  $\mathcal{P}(s,v)$  extends  $\mathcal{P}(s^*,v^*)$  and  $v^* = v - 1$  (Line 14).

**Speculation.** Replicas may speculate on a block  $B_{s_w,w}$  when it is about-to-be committed according to the prefix commit rule. That is, a replica R can speculate on a block  $B_{s_w,w}$  upon receiving a proposal  $B_{s,v}$  carrying  $\mathcal{P}(s_w,w)$  if  $B_{s_w,w}$  is from the immediately preceding slot (Line 16), which means: (1)  $s = s_w + 1$ , v = w; or (2) s = 1, v = w + 1.

**Prefix Speculation Execution.** The enforcement of the Prefix Speculation rule is similar to the basic and streamlined regime without slotting: a replica R can speculate on a bl B provided that  $\mathcal{P}(B)$  extends a certificate  $\mathcal{P}(B')$ , where B' is committed.

## 6.2 Advancing at Network Speed

Generally, leaders of BFT consensus must guarantee they extend a highest certificate that all honest replicas will accept (for liveness). A hallmark of protocols in the HotStuff family, often referred to as (optimistic) responsiveness, is allowing the protocol to advance at network speed unless there are faults. In particular, in HotStuff/HotStuff-2, the leader replacement regime ensures that (after GST), leaders learn the highest certificate without waiting for a the pre-determined maximal network delay  $\Delta$ , unless there is a fault. This feature encompasses two potential paths, happy and unhappy: a new leader waits for the highest locked certificate from the immediately preceding view (highest certificate by definition). If it obtains one, it is in a happy path and the leader can proceed to propose immediately. Otherwise, it is in an unhappy path and it waits until the previous view expires.

Streamlined HotStuff-1 with *slotting* brings a new challenge. That is,  $\mathcal{L}_v$  does not know in advance the highest slot s proposed in view v-1. Consequently, in what should be a happy path, i.e., no faults, a (good) leader may nevertheless fail to create a prepare-certificate for the last slot of his view, because the leader tries to squeeze slots

until the very end of his view. We demonstrate the influence of the such happy-unhappy scenarios with two examples.

The first example is as follows.  $\mathcal{L}_{v-1}$  proposes block  $B_{s+1,v-1}$ , but the 3f other replicas do not receive the  $B_{s+1,v-1}$  because their view-(v-1) timer expires, and their highest signature-share is for  $B_{s,v-1}$ .  $\mathcal{L}_v$ , having received  $\mathbf{n} - \mathbf{f}$  NewView messages from  $\mathcal{L}_{v-1}$  and 2f other replicas, can confirm that there is no certificate higher than  $\mathcal{P}(s,v-1)$  as it has seen signature-shares not greater than  $\mathcal{P}(s,v-1)$  from at least  $\mathbf{f}+1$  correct replicas. Then,  $\mathcal{L}_v$  stops waiting and proposes block  $B_{1,v}$  if it sees  $\mathbf{n} - \mathbf{f}$  NewView messages with signature-shares not greater than its highest certificate  $\mathcal{P}(s_{lp}, v_{lp})$  (Line 2 (1)), which also comprises the happy path.

The second example of such a happy-unhappy scenario is as follows.  $\mathcal{L}_{v-1}$  proposes block  $B_{s+1,v-1}$ . f+1 replicas receive it and send a signature-share. 2f replicas do not receive the  $B_{s+1,v-1}$  because their view-(v-1) timer expires. Hence, their highest signature-share is for  $B_{s,v-1}$ .  $\mathcal{L}_v$ , having received any subset of 3f NewView messages for view v, cannot know whether there exists one replica that knows  $\mathcal{P}(s+1,v-1)$ , because f replicas may be faulty and hide their highest signature-share. Therefore,  $\mathcal{L}_v$  must wait until either  $\mathbf{n}=3\mathbf{f}+1$  NewView messages are received (Line 2 (2)) or ShareTimer(v) (Line 2 (3)).

This scenario is somewhat rare, but breaks (optimistic) responsiveness. We are currently exploring various directions to accommodate responsiveness in Streamlined HotStuff-1 with *slotting*. Briefly, we can achieve responsiveness by (i) utilizing HotStuff in slots close(r) to view expiration, (ii) dynamically switching to HotStuff when the system experiences delays, (iii) use the first slot in each view to fix the number of slots for the view in advance; and other approaches.

#### 7 CORRECTNESS PROOFS

In this Section, we prove the *safety* and *liveness* of Streamlined HotStuff-1. We first prove the *safety* guarantee.

LEMMA 7.1. Let  $R_i$ ,  $i \in \{1, 2\}$ , be two correct replicas that executed blocks  $B_n^i$  for a given view v. If  $\mathbf{n} = 3\mathbf{f} + 1$ , then  $B_n^1 = B_n^2$ .

PROOF. Replica  $R_i$  only executes  $B_v^i$  after  $R_i$  has access to a prepare-certificate for  $B_v^i$  in accordance to Figure 5. This prepare-certificate is composed of threshold signature-shares of  $\mathbf{n}-\mathbf{f}$  replicas, which we assume cannot be compromised. Let  $S_i$  be the replicas that voted for the proposal containing block  $B_v^i$ . Let  $X_i = S_i \setminus \mathbf{f}$  be the correct replicas in  $S_i$ . As  $|S_i| = 2\mathbf{f}+1$ , we have  $|X_i| = 2\mathbf{f}+1-\mathbf{f}$ . If  $B_v^1 \neq B_v^2$ , then  $X_1$  and  $X_2$  must not overlap. Hence,  $|X_1 \cup X_2| \geq 2(2\mathbf{f}+1-\mathbf{f})$ . This simplifies to  $|X_1 \cup X_2| \geq 2\mathbf{f}+2$ , which contradicts  $\mathbf{n} = 3\mathbf{f}+1$ . Hence, we conclude  $B_v^1 = B_v^2$ .

LEMMA 7.2. If a replica R receives a certificate  $\mathcal{P}(v+1)$  that extends certificate  $\mathcal{P}(v)$ , then no certificate  $\mathcal{P}(w)$  conflicts with  $\mathcal{P}(v)$ , where view w > v, can exist.

PROOF. We know that a replica R received  $\mathcal{P}(v+1)$  that extends  $\mathcal{P}(v)$ , which is only possible if  $\mathbf{n} - \mathbf{f} = 2\mathbf{f} + 1$  replicas that set  $\mathcal{P}(v)$  as their higher known certificate also voted for  $\mathcal{P}(v+1)$ . Let's denote the  $\mathbf{f} + 1$  correct replicas from these  $\mathbf{n} - \mathbf{f}$  replicas as A. Further, certificate  $\mathcal{P}(w)$  conflicts with  $\mathcal{P}(v)$ , w > v, which implies that  $\mathcal{P}(v)$  and  $\mathcal{P}(w)$  extend the same ancestor and  $\mathcal{P}(w)$  received

support of  $\mathbf{n} - \mathbf{f} = 2\mathbf{f} + 1$  replicas. Let's denote the  $\mathbf{f} + 1$  correct replicas from these  $\mathbf{n} - \mathbf{f}$  replicas as A'. As  $w \neq v + 1$ , so w > v + 1. Moreover, any correct replica that sets  $\mathcal{P}(v)$  will not vote for a conflicting block. Thus,  $A + A' = 2\mathbf{f} + 2$ , which is more than total number of correct replicas and a contradiction.

COROLLARY 7.3. If f + 1 correct replicas speculatively execute a block  $B_v$ , then no higher conflicting certificate can commit.

PROOF. From Lemma 7.2, we implicitly get this corollary: if  $\mathbf{f} + 1$  correct replicas speculatively execute a block, then they must have set the certificate for this block as the highest known certificate, and there are not enough correct replicas in the system to vote for a conflicting certificate at a higher view.

Lemma 7.4. If a correct replica R commits a block  $B_v$ , proposed in view v, then no other block can cause it to be rollbacked.

PROOF. Assume block  $B_w$ , proposed in view w, w > v, conflicts with block  $B_v$  and another correct replica R' has committed  $B_w$ . This implies that replicas R and R' have conflicting global-ledgers. For blocks  $B_v$  and  $B_w$  to commit, R and R' must have followed the commit-rule (§5): R must have received  $\mathcal{P}(v+1)$  extending  $\mathcal{P}(v)$  and R' must have received  $\mathcal{P}(w+1)$  extending  $\mathcal{P}(w)$ . As w > v and  $w \neq v$ , so w > v+1. From Lemma 7.2, we know that once  $\mathcal{P}(v)$  and  $\mathcal{P}(v+1)$  are formed, then it is impossible to form  $\mathcal{P}(w)$ . Thus, it contradicts the fact that  $B_w$  is committed by R'.

COROLLARY 7.5. If a client receives  $\mathbf{n} - \mathbf{f}$  responses for block  $B_v$ , then no higher conflicting block can be committed.

PROOF. From Lemma 7.2 and 7.4, we implicitly get this corollary: if a client receives  $\mathbf{n} - \mathbf{f}$  responses, then at least  $\mathbf{f} + 1$  of those must have come from correct replicas. There are only two possible ways for this to happen: (1) At least  $\mathbf{n} - \mathbf{f}$  replicas speculatively executed  $B_v$  and sent reply to the client. This set of  $\mathbf{n} - \mathbf{f}$  replicas includes  $\mathbf{f} + 1$  correct replicas, and Corollary 7.3 tells us that no higher certificate will get formed. (2) At least  $\mathbf{f} + 1$  replicas executed and committed  $B_v$ , which is sufficient to guarantee that  $B_v$  cannot be rollbacked (from Lemma 7.4).

Theorem 7.6. (Safety) Streamlined HotStuff-1 guarantees a safe consensus in a system of  $n \ge 3f + 1$ .

PROOF. Using Lemma 7.1, we proved that in Streamlined HotStuff-1, no two correct replicas execute two different blocks for the same view. Further, using Lemma 7.4, we prove that a block committed by a replica will never get rollbacked, which guarantees that no two correct replicas can commit conflicting blocks. Consequently, this implies that if a replica R speculatively executes a proposal (say m) based on the Prefix Speculation rule, any proposal that m extends will not be rolled back. Moreover, if the client for m receives m - m responses, then m will definitely commit. Thus, we conclude that Streamlined HotStuff-1 guarantees safety.

Next, we prove the liveness guarantee of streamlined HotStuff-1. Like prior works [58, 83], we assume existence of GST and an appropriate view timer length  $\tau$ , which allows correct replicas to overlap in the same view after view synchronization. Such an assumption implies that the view length timer is sufficiently long to allow the leader to process NewView messages, learn the highest known

certificate, and propose a block, and for the replicas to vote. We use the notation  $v_s$  to denote the first synchronized view after GST.

Lemma 7.7. If a correct replica enters view v, then eventually all the correct replicas will enter view v.

PROOF. A correct replica exits its current view v-1 and moves to the next view v under two conditions: (1) it receives a well-formed PROPOSE message from the leader of view v-1 and post processing that message, it exits the view. (2) it receives a timeout notification from the pacemaker. Notice that at the start of a pacemaker epoch, all the replicas converge to the same view and set timers for next f leaders. Thus, if a correct replica enters view v, then eventually all the correct replicas will timeout and enter view v.

LEMMA 7.8. Assume three consecutive correct leaders  $\mathcal{L}_{v+1}$ ,  $\mathcal{L}_{v+2}$  and  $\mathcal{L}_{v+3}$ ,  $v+1 \geq v_s$ . If  $\mathcal{L}_{v+1}$  proposes a block  $B_{v+1}$  in view v+1, then all correct replicas will commit  $B_{v+1}$  in view v+3.

PROOF. Recall that the pacemaker facilitates view synchronization, which allows the leader  $\mathcal{L}_{v+1}$  to learn the highest certificate known to the correct replicas (say  $\mathcal{P}(w)$ ) once  $\mathcal{L}_{v+1}$  receives NewView messages.

 $\mathcal{L}_{v+1}$  uses this knowledge to propose block  $B_{v+1}$  that extends  $\mathcal{P}(w)$ . Each correct replica will eventually receive  $B_{v+1}$ , will set  $\mathcal{P}(w)$  as its highest known certificate (if not already set), and send a NewView message that includes a threshold signature-share in support of  $B_{v+1}$  to  $\mathcal{L}_{v+2}$ . In view v+2,  $\mathcal{L}_{v+2}$  forms  $\mathcal{P}(v+1)$  and proposes  $B_{v+2}$  extending the highest certificate  $\mathcal{P}(v+1)$ . Each correct replica will take the similar steps on receiving  $B_{v+2}$ : set  $\mathcal{P}(v+1)$  as its highest known certificate and send NewView message voting for  $B_{v+2}$  to  $\mathcal{L}_{v+3}$ . In view v+3,  $\mathcal{L}_{v+3}$  forms  $\mathcal{P}(v+2)$  and proposes  $B_{v+3}$  extending the highest certificate  $\mathcal{P}(v+2)$ . Each correct replica will eventually receive  $B_{v+3}$  and set  $\mathcal{P}(v+2)$  as its highest known certificate. Post that, all the correct replicas will commit  $B_{v+1}$ , in accordance to the *prefix commit rule*.

THEOREM 7.9. (Liveness) All correct replicas eventually commit a transaction T.

PROOF. As there are  $\mathbf{n}=3\mathbf{f}+1$  replicas in total and HotStuff-1 rotates leader in a round-robin fashion, then there is at least one set of three consecutive correct leaders:  $\mathcal{L}_{v+1}$ ,  $\mathcal{L}_{v+2}$  and  $\mathcal{L}_{v+3}$ ,  $v+1 \geq v_s$ . Thus, we can conclude from Lemma 7.8, all correct replicas will commit a transaction T proposed in view v+1.

COROLLARY 7.10. Assume two consecutive correct leaders  $\mathcal{L}_{v+1}$  and  $\mathcal{L}_{v+2}$ ,  $v+1 \geq v_s$ . If  $\mathcal{L}_{v+1}$  proposes a block  $B_{v+1}$  in view v, then  $B_{v+1}$  will eventually get committed.

PROOF. Assume we follow Lemma 7.8 to stop at two consecutive correct leaders:  $\mathcal{L}_{v+1}$  and  $\mathcal{L}_{v+2}$ . All the correct replicas will eventually receive, in view v+2, block  $B_{v+2}$  that extends  $\mathcal{P}(v+1)$  and will set  $\mathcal{P}(v+1)$  as their highest known certificate. This ensures that at no higher view a certificate that conflicts with  $\mathcal{P}(v+1)$  can exist.

Recall that after pacemaker's view synchronization, any correct leader  $\mathcal{L}_{w}$  in view  $w \geq v_{s}$  learns the highest certificate before proposing block  $B_{w}$ . Since no certificate at a higher view can conflict with  $\mathcal{P}(v+1)$ , then each block  $B_{w}$ , where view w > v+1, has certificate  $\mathcal{P}(v+1)$  as an ancestor (transitively extends). Further,

Theorem 7.9 proves that there will be at least one set of three consecutive correct leaders (say  $\mathcal{L}_w$ ,  $\mathcal{L}_{w+1}$  and  $\mathcal{L}_{w+2}$ ) and when block proposed by  $\mathcal{L}_w$  commits, all the ancestors including  $B_{v+1}$  will commit.

## 8 EVALUATION

Our evaluation aims to answer the following:

- (1) Scalability of HotStuff-1: throughput and latency with a varying number of replicas and number of transactions in a batch.
- (2) Impact of failures: leader-slowness/non-responsive leaders and tail-fork attacks.

**Setup.** We use c3.4xlarge AWS machines: 16-core Intel Xeon E5-2680 v2 (Ivy Bridge) processor, 2.8 GHz and 30 GB memory. We deploy up to 64 machines for replicas. Each experiment runs for 120 seconds. We employ *batching* in all our experiments with a default batch size of 100 and mention batch sizes in specific sections.

**Implementation.** We implement all the protocols in APACHE RESILIENTDB (incubating) [9] C++20 code with Google Protobuf v3.10.0 for serialization and NNG v1.5.2 for networking. APACHE RESILIENTDB is an optimized blockchain framework that provides APIs to implement a new consensus protocol. As threshold signature algorithms are expensive and can quickly bottleneck the computational resources, the leader sends a list of  $\mathbf{n} - \mathbf{f}$  digital signatures (from distinct replicas) as a certificate.

**Baselines.** We compare streamlined HOTSTUFF-1 against two other comparable streamlined protocols:

- (1) **HotStuff (HS-3).** The streamlined BFT consensus protocol that requires 7 half-phases to reach consensus on a client transaction (total 9 half-phases including client request and response).
- (2) **HotStuff-2 (HS-2).** Optimized HotStuff variant that requires 5 half-phases for consensus (total 7 half-phases).

As for HotStuff-1, we implement two versions of it:

- (1) **HotStuff-1.** The first streamlined BFT consensus protocol with speculative execution that requires 3 half-phases for speculative response (total 5 half-phases).
- (2) **HotStuff-1 (with Slotting).** HotStuff-1 variant with slotting; more resilient to leader slowness and tail-forking attacks.

Metrics. We focus on three metrics:

- (1) *Throughput* the maximum number of transactions per second for which the system completes consensus.
- (2) Response Latency the average duration between the time a leader proposes a client transaction to the time it sends a response for that transaction back to the client.
- (3) Client Latency the average duration between the time a client sends a transaction to the time the client receives a matching quorum of responses ( $\mathbf{f} + 1$  for HotStuff/HotStuff-2 and  $\mathbf{n} \mathbf{f}$  for HotStuff-1) for that transaction. Thus, client latency includes:  $t_1$ : the duration between the time a client sends a transaction and a replica receives the transaction;  $t_2$ : the duration between the time a transaction is received by a replica to the time it is proposed;  $t_3$ : the *response latency*; and  $t_4$ : the duration between a quorum of responses are received to when they are processed.

Durations  $t_1$ ,  $t_3$  and  $t_4$  are limited by the system message delay. Thus, we try to minimize  $t_2$ . To do so, we modify the number of inflight batches of transactions, which is the maximum number of batches of transactions a client can send before receiving sufficient

responses for any of them. We observe that the minimal numbers of inflight batches for HotStuff-1, HotStuff-2 and HotStuff are 3, 4 and 5 (a half-phase more than the total number of phases) as it allows a new leader to broadcast a new proposal while the client receives enough responses for a previous batch.

## 8.1 Scalability

First, we illustrate the scalability of HotStuff-1 against a varying number of replicas (with and without slotting), and also in comparison to HotStuff and HotStuff-2.

8.1.1 Impact of the number of replicas. In Figures 9 (a)-(c), we study various system metrics as a function of the number of replicas; we increase the number of replicas from  $\mathbf{n} = 4$  and  $\mathbf{n} = 64$  and set the number of transactions per batch to 100.

Throughput Scalability As expected, an increase in the number of replicas causes a proportional decrease in the throughput for all the protocols due to an  $O(\mathbf{n})$  increased message complexity, which decreases available bandwidth and increases the computational work at each replica. HotStuff-1 without slotting yields the same throughput as HotStuff/HotStuff-2 because the message complexity remains the same for all the streamlined variants. However, on enabling slotting in HotStuff-1, we observe a decrease in throughput because slotting unbalances load, increasing work for a leader. In particular, the slotting variant requires the leader to process multiple consecutive slots and their messages, which skews the load towards the leader for a more extended period. HotStuff-1 with slotting yields a 5.5% throughput decrease for small setups, 4 and 16 replicas, and a 11.3% decrease for large setups of 32 and 64 replicas, compared to the other protocols.

Latency Scalability An increase in the number of replicas also causes a proportional increase in the response and client latency for all the protocols due to an  $O(\mathbf{n})$  increased message complexity, which increases the time duration for a leader to collect a quorum of threshold shares and to form a certificate. Similarly, the client latency increases as the client waits for a larger quorum of messages to arrive. This indicates that HotStuff-1 clients should incur higher client latency as they must wait for  $\mathbf{f}$  more responses. However, HotStuff-1 (with or without slotting) yields lower latency because speculation guarantees an early finality confirmation. HotStuff-1 and HotStuff-1 (with slotting) yield 26.1% and 18.8% (for small setups) and 21.2% and 12.3% (for large setups) less client latency in comparison to HotStuff-2.

8.1.2 **Impact of batch size.** Next, in Figures 9 (d)-(e), we compare the three metrics for HotStuff-1 against HotStuff and HotStuff-2 as a function of the batch size; we increase the number of transactions per batch (batch size) between 100 and 10,000 and run consensus among  $\bf n=32$  replicas.

For all the protocols, an increase in batch size increases the throughput until the bandwidth or compute is saturated, increasing beyond which will cause throughput to taper off. The gain in throughput at the smaller batch sizes is due to the reduced number of consensus and processing fewer messages. At larger batches, all the protocols become compute-bounded (around batch size 5000) faster than reaching the bandwidth saturation because the gains of reduced consensus are eliminated by the overhead of proposing

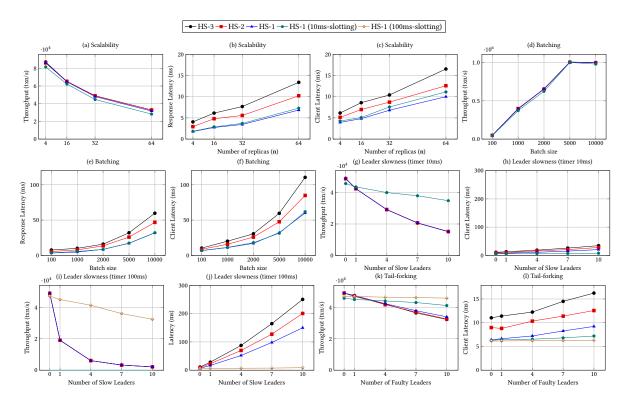


Figure 9: Impact of varying the number of replicas, batch sizes, and failures (leader slowness and tail-forking).

(for leaders) and processing (for replicas) larger batches. When the batch size is 10000, HotStuff-1 with slotting yields 1.8% lower throughput than other protocols, in comparison to a batch size of 100 with 9.8% lower throughput.

In contrast, the response and client latency increase with an increase in batch size because it takes a longer time to propose and process a larger proposal in each view.

## 8.2 Failure Resiliency

Next, we study the impact of various failures and attacks on different streamlined protocols.

**Leader slowness phenomenon** In §6, we discussed the leader slowness phenomena whereby a rational leader can delay proposing a batch of transactions until timeout (end of its rotation) in order to maximize MEV. We now study the impact of this leader slowness phenomenon on different streamlined protocols by varying the number of slow leaders from 0 to f; we set n=32 replicas and batch size to 100. Additionally, we test with two distinct timeout periods: 10 ms and 100 ms. Figure 9(g)-(j) illustrates our findings.

Unsurprisingly, the slow leaders negatively affect the throughput and client latency metrics for all the protocols but HotStuff-1 (with slotting). In the case of HotStuff-1 (with slotting), each leader has an opportunity to propose multiple batches of transactions, one per slot, which reduces the need to delay proposing a block. In fact, the larger the timeout period for a leader, the larger the number of batches it can propose. For instance, when the timeout length is  $10 \, \text{ms}$ , with  $1 \, \text{and} \, \mathbf{f} = 10 \, \text{slow}$  leaders, HotStuff (with

slotting) yields 2.8% and 1.3× higher throughput than other protocols, which changes to 1.36× and 14.57× higher at timeout length 100 ms. Similarly, for timeout length 10 ms, with 1 and f=10 slow leader, HotStuff (with slotting) yields 15.1% and 62.0% lower client latency than HotStuff-1, which changes to 58.6% and 94.1% lower at timeout length 100 ms.

**Tail-forking attack** Like the leader slowness phenomenon, the tail-forking attack aims to increase system latency by preventing correct replicas from reaching consensus on batches proposed by a correct leader (§6). In this section, by varying the number of faulty leaders from 0 to f, we illustrate the throughput and client latency metrics for all the streamlined protocols; we set n=32 replicas and batch size to 100. In our experiments, which we illustrate in Figure 9(k)-(l), a faulty leader (say view v) ignores the certificate for the proposal from the leader of the preceding view (v-1), and instead, uses the certificate for the proposal of the leader of view v-2 as an extension for its proposal.

Like earlier, the faulty leaders negatively affect the throughput and client latency metrics for all the protocols but HotStuff-1 (with slotting). In the case of HotStuff-1 (with slotting), each leader has an opportunity to propose multiple batches of transactions, while the last slot is run through a non-streamlined variant, which mitigates the impact of tail-forking attack as each correct leader proposal gets committed—a faulty leader can no longer skip.

Our results illustrate that the metrics for all of the protocols are negatively affected by the tail-forking attacks; at larger timeout values, the resiliency of HotStuff-1 (with slotting) against tail-forking attacks is visible. HotStuff-1 with *slotting* is more resilient to it,

especially when the timer length is longer. For example, when the timeout length is 10 ms, with 1 and f=10 faulty leader, HotStuff (with slotting) yield 1.6% and 10.1% lower throughput than the good case, which changes to 0.2% and 2.5% higher at timeout length 100 ms. Similarly, for timeout length 10 ms, with 1 and f=10 faulty leader, HotStuff (with slotting) yields 2.3% and 14.3% higher client latency than the good case, which changes to 0.3% and 2.9% higher at timeout length 100 ms. In contrast, the other protocols, for timeout length 10 ms, with 1 and f=10 faulty leaders yield 3.1% and 30.8% lower throughput and 3.3% and 47.3% more latency than their good case.

#### 9 RELATED WORK

Extensive literature exists on consensus and primary-backup consensus, with numerous studies (e.g., [5, 6, 8, 10, 12, 13, 15, 20, 26, 31, 38, 49, 54, 55, 64, 68, 69, 74, 79, 86]) focused on reducing communication costs and enhancing the performance and resilience of consensus systems [16, 17, 34, 37, 39, 40, 42, 48, 50, 57, 66, 70, 73, 76, 85, 87, 87].

**Speculation.** The idea of speculative execution is not new. Protocols belonging to the PBFT family have explored an *optimistic fast-path* approach to speculation [2, 31, 49]. Unfortunately, it works only in fault-free runs and requires a quadratic fallback mechanism. Several papers try to eliminate the dependence on the fast-path, but under leader failures, they also require quadratic fallback mechanisms [1, 33, 41]. Exposing the Prefix Speculation dilemma and suggesting a rule to resolve it may benefit all of these.

Rotational Leader. The HotStuff family of protocols reduces leader-replacement (view-change) communication costs to linear, enabling regular leader replacement at no additional communication cost or drop in system throughput, a challenge for protocols belonging to the PBFT family. Additionally, these protocols streamline protocol phases to double the system throughput. Among the HotStuff family, HotStuff-2 [58] achieves two-phase latency while maintaining linearity; the published HotStuff-2 algorithm is not streamlined, and streamlined HotStuff-1 contributes a streamlined variant (as well as early finality confirmation). Several other protocols have aimed to achieve two-phase streamlined and linear latency [1, 3, 28, 29, 43, 80]. However, Fast-HotStuff [43] and Jolteon [28] have quadratic complexity in view-change; AAR [3] employs expensive zero-knowledge proofs; Wendy [1] and Marlin [80] rely on a new aggregate signature construction (and are super-linear).

Parallel Proposing. Slotting is different from prior multi-proposer approaches in BFT consensus: (i) multi-leader protocols like RCC [36], MirBFT [79], and SpotLess [44], and (ii) DAG protocols [14, 25, 46, 47, 59, 75, 77, 78]. These protocols focus mostly on increasing throughput, and a majority of these protocols have a HotStuff-core. Thus, their designs are orthogonal to this paper. Any reduction in latency and elimination of leader slowness phenomena and tailforking attacks will benefit them all.

**View Synchronization.** The view-by-view paradigm of BFT protocols relies on view synchronization mechanisms to coordinate the replicas and to guarantee progress. Several solutions to the view synchronization problem have been proposed. Prior works [56, 62, 63, 82] have  $O(n^3)$  worst-case message complexity. RareSync[22]

and Lewis-Pye [51] reduce the worst-case message complexity to  $O(n^3)$  but face  $O(n\Delta)$  latency in the presence of faulty leaders. Fever [52] removes the  $O(n\Delta)$  latency but assumes a synchronous start of replicas. Lumiere [53] eliminates the need for the assumption and maintains all other properties of Fever. SpotLess [44] adopts a rapid view synchronization mechanism similar to FastSync [82], but combines view synchronization with the BFT consensus, eliminating the need for a separate sub-protocol.

Leader Slowness. The leader-slowness attack, where a rational leader delays proposing a block to maximize its MEVs, is a well-known problem in blockchains [24, 65, 67]. Prior work has illustrated that in Ethereum, for 59% of blocks, proposers have earned higher MEV rewards than block rewards [65], and any additional delay in proposing can help maximize their MEVs [72]. There are two popular solutions to tackle leader slowness: (i) Exclude any block that misses a set deadline to the main blockchain. However, a clever proposer can still delay proposing until the deadline [11]. (ii) Assign block rewards proportional to the number of attestations; a delayed block will receive fewer attestations and thus reduced block rewards [71]. However, if MEV rewards exceed total block rewards, the proposer makes a profit despite losing any block reward.

**Tail-forking attack.** As described earlier, BeeGees [30] describes the problem of tail-forking. They present an elegant solution to this problem by requiring replicas to store the proposal sent by the leader and forwarding that proposal in the future rounds. Unfortunately, re-sending these proposals over the network increases bandwdith consumption.

#### 10 CONCLUSION

The principal goal of this work has been latency reduction for client finality confirmations in streamlined BFT consensus protocols. We demonstrated that HotStuff-1 successfully lowers latency algorithmically via speculation, and furthermore, tackles leader-slowness and tail-forking attacks via slotting. Additionally, we exposed and resolved the *prefix speculation dilemma* that exists in the context of BFT protocols that employ speculation.

## REFERENCES

- Ittai Abraham, Guy Gueta, and Dahlia Malkhi. 2018. Hot-stuff the linear, optimalresilience, one-message BFT devil. CoRR, abs/1803.05069 (2018).
- [2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. https://arxiv.org/abs/1712.01367
- [3] Mark Abspoel, Thomas Attema, and Matthieu Rambaud. 2020. Malicious security comes for free in consensus with leaders. Cryptology ePrint Archive (2020).
- [4] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2011. Prime: Byzantine Replication under Attack. IEEE Trans. Depend. Secure Comput. 8, 4 (2011), 564–577. https://doi.org/10.1109/TDSC.2010.70
- [5] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In SIGMOD '21: International Conference on Management of Data. ACM, 76–88. https://doi.org/ 10.1145/3448016.3452807
- [6] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. 2024. The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation. In 21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024, Laurent Vanbever and Irene Zhang (Eds.). USENIX Association, 371-400.
- [7] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula

- Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30:1–30:15. https://doi.org/10.1145/3190508.3190538
- [8] Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. 2021. Leaderless Consensus. In 41st IEEE International Conference on Distributed Computing Systems. IEEE, 392–402. https://doi.org/10. 1109/ICDCS51616.2021.00045
- [9] Apache Software Foundation. 2023. Apache ResilientDB (Incubating). https://resilientdb.incubator.apache.org
- [10] Claudio A Ardagna, Marco Anisetti, Barbara Carminati, Ernesto Damiani, Elena Ferrari, and Christian Rondanini. 2020. A Blockchain-based Trustworthy Certification Process for Composite Services. In 2020 IEEE International Conference on Services Computing (SCC). IEEE, 422–429. https://doi.org/10.1109/SCC49832. 2020 00062
- [11] Aditya Asgaonkar. 2021. Proposer LMD Score Boosting, Ethereum Consensus-Specs. https://github.com/ethereum/consensus-specs/pull/2730
- [12] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. 2015. The Next 700 BFT Protocols. ACM Trans. Comput. Syst. 32, 4 (2015), 12:1–12:45. https://doi.org/10.1145/2658994
- [13] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. 2013. RBFT: Redundant Byzantine Fault Tolerance. In 2013 IEEE 33rd International Conference on Distributed Computing Systems. IEEE, 297–306. https://doi.org/10.1109/ICDCS. 2013.53
- [14] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2023. Mysticeti: Low-Latency DAG Consensus with Fast Commit Path. CoRR abs/2310.14821 (2023).
- [15] Christian Berger and Hans P. Reiser. 2018. Scaling Byzantine Consensus: A Broad Analysis. In Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers. ACM, 13–18. https://doi.org/10.1145/ 3284764.3284767
- [16] Adithya Bhat, Akhil Bandarupalli, Manish Nagaraj, Saurabh Bagchi, Aniket Kate, and Michael K. Reiter. 2023. EESMR: Energy Efficient BFT SMR for the masses. In Proceedings of the 24th International Middleware Conference, Middleware 2023, Bologna, Italy, December 11-15, 2023. ACM, 1-14. https://doi.org/10.1145/3590140. 3592848
- [17] Erik-Oliver Blass and Florian Kerschbaum. 2020. BOREALIS: Building Block for Sealed Bid Auctions on Blockchains. In ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security. ACM, 558–571. https: //doi.org/10.1145/3320269.3384752
- [18] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In International conference on the theory and application of cryptology and information security. Springer, 514–532.
- [19] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. CoRR abs/1807.04938 (2018).
- [20] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild (Keynote Talk). In 31st International Symposium on Distributed Computing, Vol. 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 1:1–1:16. https://doi.org/10.4230/LIPIcs.DISC.2017.1
- [21] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. ACM Trans. Comput. Syst. 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640
- [22] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. 2022. Byzantine Consensus Is Θ(n²): The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony!. In 36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs)), Vol. 246. Schloss Dagstuhl, 14:1–14:21. https://doi.org/10.4230/LIPIcs.DISC.2022.14
- [23] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation. USENIX Association, 153–168.
- [24] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2019. Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. ArXiv abs/1904.05234 (2019). https://api.semanticscholar.org/CorpusID:1212212
- [25] 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. ACM, 34-50. https://doi.org/10.1145/3492321.3519594
- [26] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. 2018. Untangling Blockchain: A Data Processing View of Blockchain Systems. *IEEE Trans. Knowl. Data Eng.* 30, 7 (2018), 1366–1385. https://doi.org/ 10.1109/TKDE.2017.2781227
- [27] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. J. ACM 35, 2 (1988), 288–323. https://doi.org/10. 1145/42282.42283

- [28] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. In *International conference on financial* cryptography and data security. Springer, 296–315.
- [29] Neil Giridharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. 2021. No-Commit Proofs: Defeating Livelock in BFT. https://eprint.iacr.org/2021/1308
- [30] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. 2023. BeeGees: Stayin' Alive in Chained BFT. In Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing (Orlando, FL, USA) (PODC '23). Association for Computing Machinery, New York, NY, USA, 233–243. https://doi.org/10.1145/3583668.3594572
- [31] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 568– 580. https://doi.org/10.1109/DSN.2019.00063
- [32] Suyash Gupta. 2021. Resilient and Scalable Architecture for Permissioned Blockchain Fabrics. Ph.D. Dissertation. University of California, Davis, USA. https://www.escholarship.org/uc/item/6901k4tj
- [33] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2021. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. In Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021, Yannis Velegrakis, Demetris Zeinalipour-Yazti, Panos K. Chrysanthis, and Francesco Guerra (Eds.). OpenProceedings.org, 301-312. https://doi.org/10.5441/002/edbt.2021.27
- [34] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2019. Brief Announcement: Revisiting Consensus Protocols through Wait-Free Parallelization. In 33rd International Symposium on Distributed Computing (DISC 2019), Vol. 146. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 44:1-44:3. https://doi.org/10.4230/LIPIcs.DISC.2019.44
- [35] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. Fault-Tolerant Distributed Transactions on Blockchain. Morgan & Claypool. https://doi.org/10. 2200/S01068ED1V01Y202012DTM065
- [36] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In 37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021. IEEE, 1392–1403. https://doi.org/10.1109/ICDE51399.2021.00124
- [37] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. Proc. VLDB Endow. 13, 6 (2020), 868–883. https://doi.org/10.14778/3380750.3380757
- [38] Suyash Gupta, Sajjad Rahnama, Erik Linsenmayer, Faisal Nawab, and Mohammad Sadoghi. 2023. Reliable Transactions in Serverless-Edge Architecture. In 39th IEEE International Conference on Data Engineering, ICDE 2023. IEEE, 301–314. https://doi.org/10.1109/ICDE55515.2023.00030
- 39] Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. 2023. Dissecting BFT Consensus: In Trusted Components we Trust!. In Proceedings of the Eighteenth European Conference on Computer Systems. ACM, 521–539. https://doi.org/10.1145/3552326.3587455
- [40] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. 2020. Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned. In 40th International Conference on Distributed Computing Systems. IEEE, 754–764. https://doi.org/10.1109/ICDCS47774.2020.00012
- [41] Jelle Hellings, Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. 2022. On the Correctness of Speculative Consensus. arXiv:2204.03552 [cs.DB] https://arxiv.org/abs/2204.03552
- [42] Heidi Howard, Fritz Alder, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Antoine Delignat-Lavaud, Cédric Fournet, Andrew Jeffery, Matthew Kerner, Fotios Kounelis, Markus A. Kuppe, Julien Maffre, Mark Russinovich, and Christoph M. Wintersteiger. 2023. Confidential Consortium Framework: Secure Multiparty Applications with Confidentiality, Integrity, and High Availability. Proc. VLDB Endow. 17, 2 (2023), 225–240. https://www.vldb.org/pvldb/vol17/p225-howard.pdf
- [43] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. 2023. Fast-HotStuff: A fast and robust BFT protocol for blockchains. IEEE Transactions on Dependable and Secure Computing (2023).
- [44] Dakai Kang, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2024. SpotLess: Concurrent Rotational Consensus Made Practical through Rapid View Synchronization. In 40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, Netherlands, May 13-17, 2024. IEEE.
- [45] Jonathan Katz and Yehuda Lindell. 2014. Introduction to Modern Cryptography (2nd ed.). Chapman and Hall/CRC.
- [46] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. In Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing. 165–175.
- [47] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. 2023. Cordial Miners: Fast and Efficient Consensus for Every Eventuality. In 37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L'Aquila, Italy (LIPIcs),

- Rotem Oshman (Ed.), Vol. 281. Schloss Dagstuhl Leibniz-Zentrum für Informatik, 26:1–26:22.
- [48] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In Proceedings of the 25th USENIX Conference on Security Symposium. USENIX, 279–296.
- [49] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2009. Zyzzyva: Speculative Byzantine Fault Tolerance. ACM Trans. Comput. Syst. 27, 4 (2009), 7:1–7:39. https://doi.org/10.1145/1658357.1658358
- [50] Lucas Kuhring, Zsolt István, Alessandro Sorniotti, and Marko Vukolić. 2021. StreamChain: Building a Low-Latency Permissioned Blockchain For Enterprise Use-Cases. In 2021 IEEE International Conference on Blockchain (Blockchain). IEEE, 130–139.
- [51] Andrew Lewis-Pye. 2022. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model. https://arxiv.org/abs/2201. 01107
- [52] Andrew Lewis-Pye and Ittai Abraham. 2023. Fever: optimal responsive view synchronisation. arXiv preprint arXiv:2301.09881 (2023).
- [53] Andrew Lewis-Pye, Dahlia Malkhi, Oded Naor, and Kartik Nayak. 2024. Lumiere: Making Optimal BFT for Partial Synchrony Practical. In Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing. 135–144.
- [54] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. 2016. XFT: Practical Fault Tolerance beyond Crashes. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. USENIX Association, USA, 485–500.
- [55] Dumitrel Loghin, Tien Tuan Anh Dinh, Aung Maw, Chen Gang, Yong Meng Teo, and Beng Chin Ooi. 2022. Blockchain Goes Green? Part II: Characterizing the Performance and Cost of Blockchains on the Cloud and at the Edge. https://arxiv.org/abs/2205.06941
- [56] Yuan Lu, Zhenliang Lu, and Qiang Tang. 2022. Bolt-Dumbo transformer: Asynchronous consensus as fast as the pipelined BFT. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. 2159–2173.
- [57] Mads Frederik Madsen, Mikkel Gaub, Malthe Ettrup Kirkbro, and Søren Debois. 2019. Transforming Byzantine Faults using a Trusted Execution Environment. In 15th European Dependable Computing Conference. IEEE, 63–70. https://doi. org/10.1109/EDCC.2019.00022
- [58] Dahlia Malkhi and Kartik Nayak. 2023. Hotstuff-2: Optimal two-phase responsive bft. Cryptology ePrint Archive (2023).
- [59] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2023. BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG. CoRR abs/2310.06335 (2023).
- [60] Jean-Philippe Martin and Lorenzo Alvisi. 2006. Fast Byzantine Consensus. IEEE Trans. Dependable Secur. Comput. 3, 3 (2006), 202–215.
- [61] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf
- [62] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. 2021. Cogsworth: Byzantine view synchronization. (2021).
- [63] Oded Naor and Idit Keidar. 2024. Expected linear round synchronization: The missing link for linear byzantine smr. Distributed Computing 37, 1 (2024), 19–33.
- [64] Faisal Nawab and Mohammad Sadoghi. 2023. Consensus in Data Management: From Distributed Commit to Blockchain. Found. Trends Databases 12, 4 (2023), 221–364. https://doi.org/10.1561/190000075
- [65] Burak Öz, Benjamin Kraner, Nicolò Vallarano, Bingle Stegmann Kruger, Florian Matthes, and Claudio Juan Tessone. 2023. Time Moves Faster When There is Nothing You Anticipate: The Role of Time in MEV Rewards. In Proceedings of the 2023 Workshop on Decentralized Finance and Security (DeFi '23). Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/ 3605768.3623563
- [66] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. 2022. RingBFT: Resilient Consensus over Sharded Ring Topology. In Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022. OpenProceedings.org, 298– 311.
- [67] Ethereum Roadmap. 2024. Proposer-Builder Separation. https://ethereum.org/en/roadmap/pbs/
- [68] Christian Rondanini, Barbara Carminati, Federico Daidone, and Elena Ferrari. 2020. Blockchain-based controlled information sharing in inter-organizational workflows. In 2020 IEEE International Conference on Services Computing (SCC). IEEE, 378–385. https://doi.org/10.1109/SCC49832.2020.00056
- [69] Pingcheng Ruan, Tien Tuan Anh Dinh, Qian Lin, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2021. LineageChain: a fine-grained, secure and efficient data provenance system for blockchains. VLDB J. 30, 1 (2021), 3–24. https://doi.org/10.1007/s00778-020-00646-1
- [70] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. 2018. EActors: Fast and flexible trusted computing using SGX. In Proceedings of the 19th International Middleware Conference, Paulo Ferreira and Liuba Shrira (Eds.). ACM, 187–200. https://doi.org/10.1145/3274808. 3274823

- [71] Caspar Schwarz-Schilling. 2022. Retroactive Proposer Rewards. https://notes.ethereum.org/@casparschwa/S1vcyXZL9
- [72] Caspar Schwarz-Schilling, Fahad Saleh, Thomas Thiery, Jennifer Pan, Nihar Shah, and Barnabé Monnot. 2023. Time Is Money: Strategic Timing Games in Proof-Of-Stake Protocols. In 5th Conference on Advances in Financial Technologies (AFT 2023) (Leibniz International Proceedings in Informatics (LIPLcs)), Vol. 282. Schloss Dagstuhl Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:17. https://doi.org/10.4230/LIPLcs.AFT.2023.30
- [73] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20). ACM, 955–970. https://doi.org/10.1145/3373376.3378469
- [74] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. 2021. BFT Protocol Forensics. In CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security. ACM, 1722–1743. https://doi.org/10.1145/3460120.3484566
- [75] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2024. Sail-fish: Towards Improving the Latency of DAG-based BFT. Cryptology ePrint Archive, Paper 2024/472.
- [76] Man-Kit Sit, Manuel Bravo, and Zsolt István. 2021. An experimental framework for improving the performance of BFT consensus for future permissioned blockchains. In DEBS '21: The 15th ACM International Conference on Distributed and Event-based Systems, Virtual Event, Italy, June 28 July 2, 2021. ACM, 55–65. https://doi.org/10.1145/3465480.3466922
- [77] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. 2023. Shoal: Improving DAG-BFT latency and robustness. arXiv preprint arXiv:2306.03058 (2023).
- [78] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022, Heng Yin, Angelos Stavrou, Cas Cremers. and Elaine Shi (Eds.). ACM. 2705–2718.
- [79] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. http://arxiv.org/abs/1906.05552
- [80] Xiao Sui, Sisi Duan, and Haibin Zhang. 2022. Marlin: Two-Phase BFT with Linearity. In 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 54-66. https://doi.org/10.1109/DSN53405.2022. 00018
- [81] Gavin Wood. 2016. Ethereum: a secure decentralised generalised transaction ledger. https://gavwood.com/paper.pdf EIP-150 revision.
- [82] Suzhen Wu, Zhanhong Tu, Yuxuan Zhou, Zuocheng Wang, Zhirong Shen, Wei Chen, Wei Wang, Weichun Wang, and Bo Mao. 2023. FASTSync: a FAST delta sync scheme for encrypted cloud storage in high-bandwidth network environments. ACM Transactions on Storage 19, 4 (2023), 1–22.
- 83] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. ACM, 347–356. https://doi.org/10.1145/3293611.3331591
- [84] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In Proceedings of the ACM Symposium on Principles of Distributed Computing. ACM, 347–356. https://doi.org/10.1145/3293611.3331591
- [85] Rui Yuan, Yubin Xia, Haibo Chen, Binyu Zang, and Jan Xie. 2018. ShadowEth: Private Smart Contract on Public Blockchain. J. Comput. Sci. Technol. 33, 3 (2018), 542–556. https://doi.org/10.1007/s11390-018-1839-y
- [86] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. 2019. GEM<sup>2</sup>-Tree: A Gas-Efficient Structure for Authenticated Range Queries in Blockchain. In 2019 IEEE 35th International Conference on Data Engineering (ICDE). IEEE, 842–853. https://doi.org/10.1109/ICDE.2019.00080
- [87] Gengrui Zhang, Fei Pan, Sofia Tijanic, and Hans-Arno Jacobsen. 2024. PrestigeBFT: Revolutionizing view changes in BFT consensus algorithms with reputation mechanisms. In 2024 IEEE 40th International Conference on Data Engineering (ICDE). IEEE, 1930–1943.

#### A APPENDIX

## A.1 Speculation Safety in Basic-HotStuff-1

Allowing replicas to speculatively execute transactions in a proposal m on receiving a certificate for m is not sufficient to guarantee safety. The following example shows that speculatively executing a block after observing a prepare-certificate violates safety.

Assume that the initial state of the system is  $\bot$  and the total number of replicas in the system are  $\mathbf{n}=3\mathbf{f}+1$ . Let's divide the  $2\mathbf{f}+1$  correct replicas into three sets: A, A', and  $A^*$ , such that  $|A|=|A'|=\mathbf{f}$  and  $|A^*|=1$ . Further, assume that the first four leaders are faulty.

- The leader of view 1,  $\mathcal{L}_1$ , proposes  $B_1$  that extends  $\mathcal{P}(\bot)$ .  $\mathbf{n} \mathbf{f}$  replicas support this proposal by sending their threshold signature-shares for  $B_1$ , which allows  $\mathcal{L}_1$  to form the prepare-certificate  $\mathcal{P}(1)$ .  $\mathcal{L}_1$  forwards this certificate to only  $\mathbf{f}$  correct replicas set A. The replicas in A speculatively execute  $B_1$  and reply to the client.
- Assume the leader of view 2,  $\mathcal{L}_2$ , ignores the highest known certificate  $\mathcal{P}(1)$  and proposes  $B_2$ , which extends  $\mathcal{P}(\bot)$  to all the replicas. Replicas in sets A' and  $A^*$  support  $B_2$ , which allows  $\mathcal{L}_2$  to form a certificate  $\mathcal{P}(2)$ .  $\mathcal{L}_2$  forwards  $\mathcal{P}(2)$  to A' only; A' replicas speculatively execute  $B_2$  and reply to the clients.
- Assume the leader of view 3,  $\mathcal{L}_3$ , ignores the highest known certificate  $\mathcal{P}(2)$  and proposes  $B_3$  that extends  $\mathcal{P}(1)$  to all the replicas. Replicas in sets A and  $A^*$  support  $B_3$ , which allows  $\mathcal{L}_3$  to form a certificate  $\mathcal{P}(3)$ .  $\mathcal{L}_3$  forwards  $\mathcal{P}(3)$  to A' only; A' replicas roll back  $B_2$ , speculatively execute  $B_3$  and its ancestor  $B_1$ .
- Assume the leader of view 4,  $\mathcal{L}_4$ , ignores the highest known certificate  $\mathcal{P}(3)$  and proposes  $B_4$  that extends  $\mathcal{P}(2)$  to all the replicas. Replicas in sets A and  $A^*$  support  $B_4$ , which allows  $\mathcal{L}_4$  to form a certificate  $\mathcal{P}(4)$ . Note: for replicas in A,  $\mathcal{P}(2)$  conflicts with their highest known certificate  $\mathcal{P}(1)$  but as  $\mathcal{P}(2)$  has a higher view number, set A replicas have to support.  $\mathcal{L}_4$  broadcasts  $\mathcal{P}(4)$  to all replicas; A replicas roll back  $B_1$ ; A' replicas roll back  $B_1$  and  $B_3$ ; all replicas speculatively execute  $B_4$  and its ancestor  $B_2$  and reply to the clients. Consequently,  $B_4$  gets set as the highest known certificate and will eventually commit.
- Unfortunately, we can have an unsafe situation where the client for  $B_1$  has received n f responses for the conflicting block  $B_1$  from A, A' and a faulty replica.

This example underscores the Prefix Speculation Dilemma: replicas vote to commit  $B_v$  with all its predecessors, but they cannot speculate on the predecessors. The Prefix Speculation rule (Definition 3.1) states that we can allow speculating only when there are no "gaps": when a replica votes to commit  $B_v$ , it can speculate on  $B_v$  only if  $B_v$  extends a committed block  $B_w$ . Hence, there are no gaps, and we speculate only on the block in the current view, which is safe. From the existing literature on speculative consensus protocols, we note that Zyzzyva's practice of requiring replicas to send speculation results *carrying a view number* and requiring clients not to mix speculation results from different views can be handy.

## A.2 Rollback is Necessary

Providing early finality confirmation is speculative. If a conflicting certificate is formed at a higher view, the local-ledger needs to be rollbacked. We illustrate this in the following scenario.

Assume that the initial state of the system is  $\bot$ . The leader of view 1,  $\mathcal{L}_1$ , proposes m that extends  $\mathcal{P}(\bot)$ .  $\mathbf{n} - \mathbf{f}$  replicas support this proposal by sending their threshold signature-shares for m, which allows  $\mathcal{L}_1$  to form the prepare-certificate  $\mathcal{P}(1)$ .  $\mathcal{L}_1$  forwards this certificate to  $\mathbf{f}$  correct replicas; let us denote this set of replicas A. The replicas in A speculatively execute  $B_1$  in m and reply to the clients. Assume the leader of view 2,  $\mathcal{L}_2$ , is also faulty; it ignores the highest locked certificate  $\mathcal{P}(1)$  and proposes m' that extends  $\mathcal{P}(\bot)$  to all the replicas. All but set A replicas (say set A') support m', which allows  $\mathcal{L}_2$  to form a certificate  $\mathcal{P}(2)$  as there are at least  $\mathbf{n} - \mathbf{f}$  replicas in A'.  $\mathcal{L}_2$  broadcasts  $\mathcal{P}(2)$  to all the replicas. On receiving  $\mathcal{P}(2)$ , set A replicas will rollback their local-ledger as  $\mathcal{P}(2)$  is formed at a higher view than  $\mathcal{P}(1)$ . Post this, all the correct replicas speculatively execute  $B_2$  and reply to the clients, which mark transactions in  $B_2$  as complete (received  $\mathbf{n} - \mathbf{f}$  responses).

# A.3 Speculation Safety in Streamlined HotStuff-1

The following example shows that speculatively executing a block after observing a two-chain of prepare-certificates in streamlined HotStuff-1 violates safety.

Assume that the initial state of the system is  $\bot$  and the total number of replicas in the system are  $\mathbf{n} = 3\mathbf{f} + 1$ . Let's divide the  $2\mathbf{f} + 1$  correct replicas into three sets: A, A', and  $A^*$ , such that  $|A| = |A'| = \mathbf{f}$  and  $|A^*| = 1$ .

- The leader of view 1,  $\mathcal{L}_1$ , proposes  $B_1$  that extends  $\mathcal{P}(\bot)$ .  $\mathbf{n} \mathbf{f}$  replicas support this proposal by sending their threshold signature-shares for  $B_1$  to  $\mathcal{L}_2$ , leader of view 2, which forms the prepare-certificate  $\mathcal{P}(1)$ . Assume the leader of view 2,  $\mathcal{L}_2$ , proposes  $B_2$  that extends  $\mathcal{P}(1)$  and forwards this certificate to only  $\mathbf{f}$  correct replicas set A. The replicas in A speculatively execute  $B_1$  and reply to the client (two-chain of certificates:  $\mathcal{P}(\bot)$  and  $\mathcal{P}(1)$ ).
- Assume the leader of view 3,  $\mathcal{L}_3$  propose  $B_3$  that extends  $\mathcal{P}(\bot)$  and send to replicas in set Replicas in sets A' and  $A^*$  support  $B_3$ , which allows  $\mathcal{L}_4$ , leader of view 4, to form a certificate  $\mathcal{P}(3)$ . Assume  $\mathcal{L}_4$  propose  $B_4$  that extends  $\mathcal{P}(3)$  and send to replicas in set A'; A' replicas speculatively execute  $B_3$  and reply to the clients.
- Assume the leader of view 5,  $\mathcal{L}_5$  ignores the highest known certificate  $\mathcal{P}(3)$  and propose  $B_5$  that extends  $\mathcal{P}(1)$  to all the replicas. Replicas in sets A and  $A*B_5$ , which allows  $\mathcal{L}_6$ , leader of view 6, to form a certificate  $\mathcal{P}(5)$ .  $\mathcal{L}_6$  forwards  $\mathcal{P}(5)$  to A' only; A' replicas roll back  $B_3$ , speculatively execute  $B_5$  and its ancestor  $B_1$ .
- Assume the leader of view 7,  $\mathcal{L}_7$ , ignores the highest known certificate  $\mathcal{P}(5)$  and proposes  $B_7$  that extends  $\mathcal{P}(3)$  to all the replicas. Replicas in sets A and  $A^*$  support  $B_7$ , which allows  $\mathcal{L}_8$  to form a certificate  $\mathcal{P}(7)$ . Note: for replicas in A,  $\mathcal{P}(3)$  conflicts with their highest known certificate  $\mathcal{P}(1)$  but as  $\mathcal{P}(3)$  has a higher view number, set A replicas have to support.  $\mathcal{L}_8$  broadcasts  $\mathcal{P}(7)$  to all replicas; A replicas roll back  $B_1$ ; A' replicas roll back  $B_1$  and  $B_5$ ; all replicas speculatively execute  $B_7$  and its ancestor  $B_3$  and reply to the clients. Consequently,  $\mathcal{P}(7)$  gets set as the highest known certificate and will eventually commit.

• Unfortunately, we can have an unsafe situation where the client for  $B_1$  has received n-f responses for the conflicting block  $B_1$  from A, A' and a faulty replica.

The problem is there is a **gap**. The replicas can vote to commit on  $B_5$ , but they cannot speculate on  $B_1$ . It is "too late" for them to vote

or speculate on ancestors, it would be unsafe. The replicas must not execute/speculate on  $B_1$ . We still need a *no-gap* rule (*prefix-commit*) here, allowing speculation only one block at a time, provided it extends a committed predecessor