

HOTSTUFF-1: Linear Consensus with One-Phase Speculation

DAKAI KANG, University of California, Davis, USA

SUYASH GUPTA, University of Oregon, USA

DAHLIA MALKHI, University of California, Santa Barbara, USA

MOHAMMAD SADOOGHI, University of California, Davis, USA

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. Furthermore, HOTSTUFF-1 incorporates an incentive-compatible leader rotation design that motivates leaders to propose transactions promptly. HOTSTUFF-1 achieves a reduction of two network hops by *speculatively* sending clients early finality confirmations, after one phase of the protocol. Introducing speculation into streamlined protocols is challenging because, unlike *stable-leader* protocols, these protocols cannot stop the consensus and recover from failures. Thus, we identify *prefix speculation dilemma* in the context of streamlined protocols; HOTSTUFF-1 is the first streamlined protocol to resolve it. HOTSTUFF-1 embodies an additional mechanism, *slotting*, that thwarts delays caused by (1) rationally-incentivized leaders and (2) malicious leaders inclined to sabotage other's progress. The slotting mechanism allows leaders to dynamically drive as many decisions as allowed by network transmission delays before view timers expire, thus mitigating both threats.

CCS Concepts: • **Information systems** → **Distributed database transactions**.

Additional Key Words and Phrases: Byzantine Fault Tolerance, Consensus Protocol, Speculative Execution, Low-Latency Consensus, Blockchain, Distributed Database Systems

ACM Reference Format:

Dakai Kang, Suyash Gupta, Dahlia Malkhi, and Mohammad Sadoghi. 2025. HOTSTUFF-1: Linear Consensus with One-Phase Speculation. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 171 (June 2025), 29 pages. <https://doi.org/10.1145/3725308>

1 Introduction

This paper introduces HOTSTUFF-1, a BFT consensus protocol designed to reduce latency while simultaneously maintaining scalability. HOTSTUFF-1 is primarily motivated by blockchains and online platforms that support digital asset payments and marketplaces [10, 26, 84]. These 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 these systems, especially financial platforms, response latency is crucial for user engagement and satisfaction. Moreover, the demands for low response latency are posed not only by the market but also by regulation. A manuscript detailing regulatory technical requirements for Financial Market Infrastructure (FMI) states 12 key standards for operating an FMI, among which are performance requirements such as meeting peak throughput demand and timely responsiveness [2].

In this paper, we are interested in BFT consensus protocols for a *partially-synchronous* setting, due to their safety against temporary network delays. Pioneering BFT consensus protocols belonging to the PBFT family [29, 50] employ a *stable-leader* design, where one replica designated as the

Authors' Contact Information: Dakai Kang, dakang@ucdavis.edu, University of California, Davis, USA; Suyash Gupta, suyash@uoregon.edu, University of Oregon, USA; Dahlia Malkhi, dahliamalkhi@ucsb.edu, University of California, Santa Barbara, USA; Mohammad Sadoghi, msadoghi@ucdavis.edu, University of California, Davis, USA.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2836-6573/2025/6-ART171

<https://doi.org/10.1145/3725308>

leader initiates a two-phase consensus algorithm that determines the ledger. Unfortunately, the stable-leader design has some drawbacks.

D1: a dedicated leader increases *ensorship opportunities*, as the leader decides what transactions to propose [109].

D2: when the leader fails, these protocols switch to a *view-change* algorithm that incurs quadratic communication complexity to replace the leader (or change the view) and drops the system throughput to zero, as consensus on new transactions can start only after the view-change [6, 32].

D3: it *inhibits load and reward balancing* among the replicas [51].

D4: a malicious leader can *keep the system throughput at the lowest level* and prevent detection by proposing transactions just before the timeout period [6, 17, 32].

Some recent protocols that follow the stable-leader design attempt to solve **D3** and **D4** by requiring all the replicas to act as the leader and/or track the leader's performance [6, 17, 32, 51, 70, 102]. However, these works require several redundant rounds of consensus that track the leader's performance (e.g. RBFT [17] and Fairledger [70]) and face collusion attacks by multiple malicious leaders (e.g. MirBFT [102] and RCC [51]).

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 [25]. Later, HotStuff [109] reduced view-change communication costs to linear, and additionally *streamlined* protocol phases to (at least) double throughput (solving **D1** to **D4**). 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 the following three additional challenges:

D5: Increased latency. Despite recent improvements [79], streamlined protocols incur higher latency than the stable-leader protocols that employ optimizations like *speculative-execution* [45, 48].

D6: Leader-slowness phenomenon. In blockchain systems, regular leader replacement creates an undesirable incentive structure: a leader may be inclined to delay proposing a block of transactions as close as possible to the end of its view expiration period in order to pick the transactions that offer the highest fees. Similarly, block-builders participating in a proposer-builder auction will wait as long as possible to maximize MEV (maximal extractable value) exploits [34, 89, 91]. Thus, rational leaders/builders may slow down progress and cause clients to suffer increased latency.

D7: Tail-forking attack. BeeGees [44] exposed another vulnerability of streamlined protocols, where faulty leaders prevent proposals by correct leaders from being committed unless there are consecutive correct leaders. This attack surfaces when faulty leaders are interjected between correct 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, stable-leader protocols yield optimal latency under no-failure cases through speculative execution and do not face **D5** to **D7**. However, they have yet to solve **D1** and **D2**, and solving **D3** and **D4** introduces new challenges. On the other hand, streamlined protocols resolve **D1** to **D4** but have yet to solve **D5** to **D7**.

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

- (1) A novel algorithmic core that combines regular leader rotation with linear communication, streamlining and 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* algorithm that provides each leader with multiple slots to propose transactions. HOTSTUFF-1 uses slotting to maintain consistent high performance by mitigating the impacts of leader-slowness and tail-forking.

Early Finality Confirmation through Speculation. The notion of applying speculative execution to BFT protocols is not new. In his PhD thesis [28], Miguel Castro presented the idea of applying *tentative execution* to PBFT, which was later expanded/evaluated by PoE [48]. Several other flavors of speculative execution also exist (Zyzyva [67] and SBFT [45]). These papers illustrate that speculative execution can reduce the latency of BFT consensus in the no-failure case. Unfortunately, applying speculative execution to streamlined protocols is not a straightforward extension.

These stable-leader protocols **stop** speculative execution during the recovery/view change phases because they need to run an explicit view-change protocol (**D2**). At the end of the view-change protocol, all replicas start the new view when they receive from the new leader a *state*. This state starts from the last agreed-upon checkpoint, and for each sequence number that some replica claims to have observed since the last checkpoint, this state includes a prepare-certificate (if available) or a proposal from the previous leader.¹ However, before a replica can add any of these sequence numbers/proposals to its log, the leader needs to rerun consensus on each of them.

Streamlined protocols **do not have** the option of stopping consensus and rerunning consensus on past transactions, which makes introducing speculation challenging. Thus, we identify the existence of a conundrum when applying speculation to the streamlined protocols; we term this conundrum as the *prefix speculation dilemma*. HOTSTUFF-1 is the first streamlined protocol to employ speculative execution and resolve this conundrum by dictating when it is safe for a replica to speculatively execute a proposal.

Consequently, HOTSTUFF-1 treats clients as first-class citizens of consensus by serving them with *early finality confirmation*. HOTSTUFF-1 builds streamlining and speculation over HOTSTUFF-2 [79]. Unlike HOTSTUFF-2, which forces replicas to wait until they learn whether a transaction has committed, HOTSTUFF-1 allows replicas to send commit-votes on transactions directly to clients when a transaction is prepared and highly likely to commit, which also allows replicas to speculate on the execution results and send responses to clients. On collecting responses from a quorum of $n - f$ replicas, clients learn two things at once: a commit decision and its execution result, which enables an early finality confirmation. Thus, HOTSTUFF-1 meets the challenges **D1** to **D5**.

Low latency through slotting. HOTSTUFF-1 resolves a subset of the challenges we listed earlier in this section, but challenges like leader slowness (**D6**) and tail-forking attacks (**D7**) remain. Therefore, we 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. However, fixing the number of slots per leader/view does not eliminate the slowness attack; a fast leader will slow down its last slot. Therefore, we devise an *adaptive* slotting mechanism that allows a leader to propose as many slots as it can during the time span allotted to its view. Permitting adaptive slotting in a streamlined consensus protocol unravels a new challenge: how can the subsequent leader determine if it has received the certificates corresponding to the last slot of the preceding leader? We introduce the notion of *trusted/distrusted previous leaders* to enable a correct leader to propose its first slot at the network speed between itself and the previous leader if the previous leader is correct.

Resilience to tail-forking attacks. HOTSTUFF-1 with slotting guarantees that in each view v , if $\mathcal{L}_v$ proposed at least two slots, at most one could remain uncertified, and it could only be tail-forked if fewer than $f + 1$ correct replicas voted for it. This is achieved via *carry* blocks and a

¹Alternatively, if the new leader does not have access to any prepare-certificate for a sequence number, it can leave that sequence number as empty [29].

dual-certificate mechanism: *New-View* and *New-Slot* certificates, which enforce the slot's inclusion in the well-formed first-slot proposal sent by the next leader.

We illustrate the practicality of our design by implementing HOTSTUFF-1 (with and without slotting) in APACHE RESILIENTDB (incubating) [12] and evaluating it against two baselines: HOTSTUFF and HOTSTUFF-2. Our results affirm that HOTSTUFF-1 yields lower latency than the baselines; in the no-failure case, HOTSTUFF-1 (with and without slotting) yields up to 41.5% and 24.2% lower latency. Additionally, we illustrate the resistance of HOTSTUFF-1 (with slotting) against leader-slowness and tail-forking attacks. In summary, we make the following contributions:

- (1) We introduce HOTSTUFF-1, the first speculative, streamlined and linear BFT consensus protocol that serves clients with early finality confirmations for their transactions.
- (2) We expose a prefix speculation dilemma that exists in the context of streamlined BFT protocols that employ speculation and present a solution tailored for HOTSTUFF-1.
- (3) We introduce slotting in HOTSTUFF-1 to mitigate leader-slowness and tail-forking attacks. Our slotting mechanism is adaptive, yet guarantees no delay for subsequent leaders.

2 Background and System Model

Modern databases require replication to guarantee availability to their clients; consensus protocols help keep these replicas consistent [69, 88]. As consensus can quickly bottleneck system performance, a large body of existing work attempts to optimize these protocols [106]. A majority of existing databases [33, 58, 105] employ *crash fault-tolerant* consensus protocols to guarantee consistent replication despite crash failures [69, 88]. This crash-failure threat model is sufficient for these databases as they are managed by a single organization. In this paper, we focus on designing efficient Byzantine Fault-Tolerant (BFT) consensus protocols that can guard against arbitrary malicious failures. Such protocols are necessary for databases managed by multiple parties; commonly used in financial trading and blockchain applications [7, 12, 104].

We assume the system model adopted by existing partially synchronous BFT consensus protocols [29, 45, 48, 67, 109]. We assume a system of n replicas, of which at most f are faulty (malicious or crash-failed), and the remaining $n - f$ replicas are correct; $n \geq 3f + 1$. 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 $[1, n]$ using function $\text{id}(R)$.

Authenticated communication: each client/replica uses digital signatures to sign a message [63]. Additionally, replicas make use of the BLS threshold signature scheme [23] to form (n, t) threshold signatures. Each replica R has access to a private signature key, which it uses to create a signature share δ_R . An aggregator needs only t shares out of 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 the existence of a collision-resistant hash function $H(x)$, where it is impossible to find a value x' , such that $H(x) = H(x')$ [63].

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 [39] where there is a known bound Δ on message transmission delays, such that after an unknown time called *GST* all transmissions arrive at their destinations within Δ bounds.

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

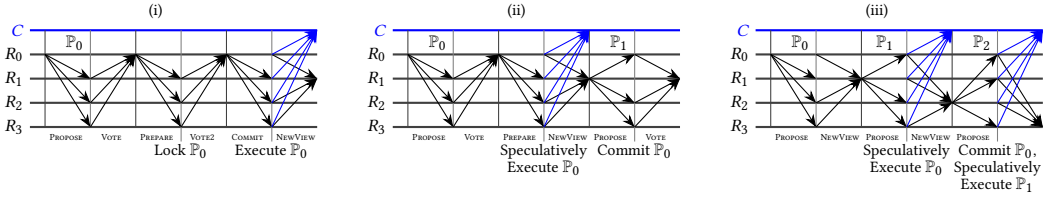


Fig. 1. Workflows of (i) BASIC HOTSTUFF-2, (ii) BASIC HOTSTUFF-1, and (iii) STREAMLINED HOTSTUFF-1

in sequential order. There are two requirements; *safety* is required under asynchrony and *liveness* is required under 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 Speculation in Streamlined Protocols

Our primary goal is to reduce the latency for partially-synchronous streamlined consensus protocols. That is, we aim to bridge the gap between the latency of streamlined protocols and optimized stable-leader consensus protocols without losing a vital tenet: linearity. An additional goal of this work is to mitigate the slowness attacks and tail-forking attacks 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) resilience to tail-forking attack.

In the rest of this section, we discuss HOTSTUFF-1 (speculation) and defer discussion on slotting until §6. To illustrate the challenges in introducing speculation to streamlined consensus protocols, we first briefly recap the skeleton of the HOTSTUFF-2 [79] protocol.

Recap of HotStuff-2

HOTSTUFF-2 optimizes HOTSTUFF by reducing commit latency by one phase (or two half-phases). 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-a-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 $n - f$ 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 $n - f$ replicas. The second is a *commit-certificate*, a certificate-of-certificate, guaranteeing that $n - f$ replicas have received the prepare-certificate, and despite any 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 send client responses).

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. HOTSTUFF-1 requires replicas to employ speculative execution to serve clients with early finality confirmations.

Rather than requiring replicas to wait until they learn whether a transaction *has committed*, HOTSTUFF-1 allows replicas to speculate precisely when a transaction is prepared and highly likely to 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 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 commit-votes and the result of executing T , which enables an early finality confirmation. When a client receives responses from a quorum of $n - f$ replicas, it learns two things: a transaction has been committed, and the execution result has been computed in advance. Safety follows from the commit-safety of HOTSTUFF-2 because a client can determine if a commit-certificate will form.

In a non-speculative protocol, a client needs to collect only $f + 1$ execution responses to determine the finality because correct replicas execute a transaction once the commit decision is reached; response from just one correct replica guarantees commitment. However, in HOTSTUFF-1, clients need to collect $n - f$ responses because $f + 1$ speculative responses only guarantee that one correct replica prepares the transaction. Commitment is guaranteed only when at least $f + 1$ correct replicas prepare the transaction. Thus, a client learns that a transaction will finalize only upon collecting $n - f$ responses. Figure 1 (ii) depicts our HOTSTUFF-1 protocol.

Although clients of HOTSTUFF-1 wait for f additional messages (temporarily increasing memory footprint) compared to clients of HOTSTUFF and HOTSTUFF-2, we argue that HOTSTUFF-1 clients do not incur higher latency because they receive early finality confirmations. In §7, we conduct several experiments to validate this claim. Moreover, a client can delay verifying and processing these additional responses as long as necessary while prioritizing other tasks. Such a delay would not impact the latency of HOTSTUFF-1 because replicas do not wait for any input from the client.

The Prefix Speculation Dilemma.

In HOTSTUFF-1, when a client receives a quorum of responses for a transaction T , it learns that T will be committed and that finality has been reached for appending T to the ledger in sequence order. This decision also commits all transactions preceding T , and the result of executing T reflects processing the *full prefix* of transactions up to and including T . However, responses for T **must not** be combined with those for preceding transactions to form a quorum. That is, say T succeeds an earlier transaction T' in sequence order, $T' \prec T$. The commit-votes (speculative responses) of T must not be used as commit-votes of T' in forming a commit-decision on T' .

This brings forth a challenging dilemma with respect to speculation²: the responses from T represent the execution of a full prefix ending with T . When a replica R speculatively executes T , it must execute all transactions that precede it. However, if R did not commit the preceding transaction T' prior to executing T , it **must not** send responses for T to clients because these responses represent commit-votes. Otherwise, clients can mistakenly combine commit-votes from a partial quorum on T' with commit-votes from another partial quorum on T and assume that a decision has been reached on T' . On the other hand, R must ensure there is “**no view gap**” between the view in which T is prepared and its current view, to prevent speculative execution on a proposal that might be superseded by a higher certificate that is formed in the gap and unknown to R .

²See Appendix A.1 in our extended report for a detailed explanation of why the dilemma breaks *safety* [61].

Note on Speculation in Stable-Leader Protocols.

As stated in the introduction, the notion of applying speculative execution to BFT protocols is not new [28, 45, 48, 67]. However, we argue that applying speculative execution to streamlined protocols is not a straightforward extension.

These stable-leader protocols stop speculative execution during the view change phases because they need to run an explicit view-change protocol. At the end of the view-change protocol, all the replicas start the new view when they receive from the new leader a *state*. This state starts from the last agreed-upon checkpoint, and for each sequence number that some replica claims to have observed since the last checkpoint, this state includes a prepare-certificate (if available) or a proposal from the previous leader. However, before a replica can add any of these sequence numbers/proposals to their log, the leader needs to re-run consensus on each of them.

Even though stable-leader protocols require their clients to not combine votes on a transaction across views, the ability to stop consensus, change views, and re-run consensus on past transactions ensures that neither there is a situation where a replica is executing a transaction T but is yet to commit a preceding transaction T' nor there is a “view gap”.

Tackling Prefix Speculation Dilemma.

Streamlined protocols **do not have** the option of stopping the consensus and re-running consensus on past transactions. Thus, we state the following two rules to tackle the prefix speculation dilemma in streamlined protocols:

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

Definition 3.2. No-Gap Rule. A replica R , currently in view v , may speculatively execute a transaction T only if T is proposed in view $v - 1$ and a prepare-certificate of T is formed in view v .

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 T will commit. Instead, each replica maintains a *local-ledger*, where it marks T prepared and executed. If in a succeeding view, R is about to speculatively execute a transaction T' that conflicts with T (See Definition 4.4), then R must perform a *rollback* operation in the local-ledger. R can observe that at least $n - f$ replicas prepared T' and then T cannot commit. Specifically, the replica should now fetch the transaction T' from other replicas, erase T from its local-ledger, execute T' , add an entry for T' to its global-ledger, and respond to the client. We discuss this in more detail in §4.2.

4 Speculative Core

We first describe the variant of basic (non-streamlined) HOTSTUFF-1 variant; in §5, we describe the streamlined HOTSTUFF-1.

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 including T and sends it to one replica.

Client Response. When a client c receives identical RESPONSE messages from $n - f$ 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.

Local state (replica R) :

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

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

- 6: **event** Upon `PACEMAKER.ENTREVIEW( $v$ )` **do**
- 7: Wait until received $n - f$ `NEWVIEW` messages for view v
- 8: Wait until received $\mathcal{P}(v - 1)$ **or** received n `NEWVIEW` messages **or** `PACEMAKER.SHARETIMER( $v$ )`
- 9: Let B_v be a block of client transaction yet to be proposed
- 10: Broadcast $m = \langle \text{PROPOSE}, B_v, v, \mathcal{P}(v_{lp}), \mathcal{C}(v_{lc}) \rangle_{\mathcal{L}_v}$
- 11: **event** Received $n - f$ `NEWVIEW` messages with shares for $\mathcal{P}(v - 1)$ **do**
- 12: $\mathcal{C}(v - 1) \leftarrow \text{CREATETHRESHOLDSIGN}(n - f \text{ distinct } \delta_R^{\mathcal{C}} \text{ shares})$
- 13: **event** Received $n - f$ `PROPOSEVOTE` messages **do**
- 14: $\mathcal{P}(v) \leftarrow \text{CREATETHRESHOLDSIGN}(n - f \text{ distinct } \delta_R^{\mathcal{P}} \text{ shares})$
- 15: Broadcast $\langle \text{PREPARE}, v, \mathcal{P}(v) \rangle_{\mathcal{L}_v}$

Backup role (running at each replica R (including leader)) :

- 16: **event** Received $\langle \text{PROPOSE}, B_v, \mathcal{P}(w), \mathcal{C}(x) \rangle_{\mathcal{L}_v}$ **do**
- 17: Execute all transactions up to (incl.) B_x , add result to global-ledger and respond to clients $\triangleright$ *traditional-commit rule*
- 18: **if** $w \geq v_{lp}$ $\triangleright$ *vote to prepare B_v* **then**
- 19: $\delta_R^{\mathcal{P}} \leftarrow \text{CREATETHRESHOLD SHARE}(\mathcal{P}(w), v, \text{Hash}(B_v))$
- 20: Send $\langle \text{PROPOSEVOTE}, v, \delta_R^{\mathcal{P}} \rangle_R$ to $\mathcal{L}_v$
- 21: **event** Received $\langle \text{PREPARE}, v, \mathcal{P}(v) \rangle_{\mathcal{L}_v}$ **do**
- 22: **if** $\mathcal{P}(v)$ extends $\mathcal{P}(v - 1)$ $\triangleright$ *prefix-commit rule* **then**
- 23: Execute all transactions up to (incl.) B_{v-1} , add result to global-ledger and respond to clients
- 24: **if** predecessor of B_v is in global-ledger $\triangleright$ *Prefix Speculation rule* **then**
- 25: **if** local-ledger state conflicts with B_v **then**
- 26: Roll local-ledger back to the common ancestor
- 27: Execute all transactions in B_v speculatively, add result to local-ledger and send client a response $\triangleright$ *speculatively execute B_v*
- 28: $\delta_R^{\mathcal{C}} \leftarrow \text{CREATETHRESHOLD SHARE}(\mathcal{P}(v)) \triangleright$ *vote to commit B_v*
- 29: Send $\langle \text{NEWVIEW}, v + 1, \mathcal{P}(v), \delta_R^{\mathcal{C}} \rangle_R$ to $\mathcal{L}_{v+1}$
- 30: Call `EXITVIEW()`
- 31: **event** Upon timeout **do**
- 32: Send $\langle \text{NEWVIEW}, v + 1, \mathcal{P}(v_{lp}), \perp \rangle_R$ to $\mathcal{L}_{v+1}$.
- 33: Call `EXITVIEW()`
- 34: **function** `EXITVIEW()` **do**
- 35: $v \leftarrow v + 1$. $\triangleright$ *disable voting and speculative execution for view v*
- 36: Call `PACEMAKER.COMPLETEDVIEW()`

Fig. 2. Basic HOTSTUFF-1.

Replica pseudocode. In Figure 2, we present the pseudo-code for basic HOTSTUFF-1. Prior to describing the algorithm in detail, we lay down some useful definitions.

Definition 4.1. Prepare and Commit Certificates. A prepare-certificate $\mathcal{P}(v)$ for a proposal m aggregates $\mathbf{n} - \mathbf{f}$ threshold signature-shares for m in view v . 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 4.2. Highest Known Certificate. A certificate $\mathcal{P}(v_{lp})$ for view v_{lp} is the highest prepare-certificate, known to replica R . For brevity, we omit from the code explicitly updating v_{lp} every time R learns a new certificate.

Definition 4.3. 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 $\mathcal{P}(w)$. 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 $\mathcal{P}(w)$.

Definition 4.4. 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 $v = \text{id}(R) \bmod \mathbf{n}$ —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 $\mathcal{P}(v - 1)$, the leader forms a commit-certificate $C(v - 1)$ (Line 12) and updates $C(v_{lc})$. After learning the highest certificate across all correct replicas (Line 8, see further explanations in §4.2), the leader aggregates client transactions (yet to be proposed) into a block B_v and creates a PROPOSE message m that includes the view number v , B_v , $\mathcal{P}(v_{lp})$, and $C(v_{lc})$. Then, $\mathcal{L}_v$ broadcasts m to all replicas (Lines 9-10). *Note.* The PROPOSE message for view 0, the genesis view, extends a hard-coded certificate that all replicas assume to be valid.

ProposeVote. On receiving a PROPOSE message m from $\mathcal{L}_v$ (Line 16), a replica R checks if the prepare certificate $\mathcal{P}(w)$ in m is not lower than its highest prepare-certificate $\mathcal{P}(v_{lp})$, i.e., $w \geq v_{lp}$.

If $w > v_{lp}$, then R updates its v_{lp} to w , sets $\mathcal{P}(w)$ as the highest known prepare-certificate and fetches the block corresponding to $\mathcal{P}(w)$ from other replicas. (§4.2).

If $w \geq v_{lp}$, R creates a PROPOSEVOTE message, which includes a threshold signature-share $\delta_R^{\mathcal{P}}$ for m , and sends this message to $\mathcal{L}_v$ (Lines 16-20). Otherwise, R ignores the message.

Prepare. When $\mathcal{L}_v$ receives $\mathbf{n} - \mathbf{f}$ well-formed PROPOSEVOTE messages for its proposal m , it combines their signature shares into a threshold signature to create a prepare-certificate $\mathcal{P}(v)$ (Lines 13-14). Then, $\mathcal{L}_v$ creates a PREPARE message including $\mathcal{P}(v)$ and broadcasts it (Line 15).

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

If B_v 's predecessor is already in the global-ledger (i.e., meets the Prefix Speculation rule) and B_v was prepared in view v (i.e., meets the No Gap rule³), R does the following (Lines 21-27):

- (1) *Speculatively executes* the transactions in block B_v of m .
- (2) Send speculative responses with execution results to the respective clients.
- (3) Adds result of executing B_v to its local-ledger.

ExitView and NewView. A replica R exits 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

³We defined No Gap rule for streamlined protocols in § 3. However, it also implicitly applies to the non-streamlined versions: A replica R , currently in PROPOSEVOTE phase of view v , may speculatively execute a transaction T only if T was proposed in the preceding PROPOSE phase of view v and a prepare-certificate of T is formed in view v .

a **NEWVIEW** message, which includes $\mathcal{P}(v_{lp})$, and forwards it to the leader $\mathcal{L}_{v+1}$ of view $v + 1$. It then invokes the pacemaker to orchestrate view-synchronization as needed (Line 36).

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 4.5. 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} .

Definition 4.6. Prefix Commit Rule. A replica marks a block B_{v-1} as committed when it receives a prepare-certificate $\mathcal{P}(v)$ 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 creates a threshold share (δ_R^C) for the prepare-certificate and forwards this threshold share with the **NEWVIEW** message to the leader of the next view (Lines 28-29). Next, each replica calls the **EXITVIEW** procedure. On receiving $n - f$ threshold shares for the same prepare-certificate, the leader of the next view combines them into a commit-certificate (Line 12) and forwards it to all the replicas. Upon receiving a commit-certificate $C(v)$, a replica R adds the block B_v to the global-ledger and marks it committed (Line 17). *Note:* on receiving the commit-certificate, R sends a response 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, which we will expand on in § 4.2.

4.2 Failures and Recovery Design

A malicious replica can impact the consensus in various ways if it is the leader of an ongoing view: (1) drop, delay, or prevent sending messages and/or certificates to prevent replicas from making progress, and (2) equivocate by creating 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 of the current view (say v) has failed and thus sends a **NEWVIEW** message to the leader of view $v + 1$. Post this, R calls the **EXITVIEW** procedure to move to the next view (Lines 31-36).

Lack of certificates from 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 preceding leader. If it extends some lower certificate, its new proposal will get ignored by correct replicas that received $\mathcal{P}(v-1)$. To ensure that the new proposal will be voted 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), after *GST*, it is guaranteed that by *PACEMAKER.SHARETIMER*(v), which is 3Δ after $\mathcal{L}_v$ enters view v , $\mathcal{L}_v$ will receive **NEWVIEW** messages including known certificates from all correct replicas. Thus, if $\mathcal{L}_v$ did not receive $\mathcal{P}(v-1)$, it should wait until either it received n **NEWVIEW** messages or *PACEMAKER.SHARETIMER*(v) (Line 8).

Conflict Resolution: Rollback

When a replica R receives a prepare-certificate $\mathcal{P}(v)$, **HOTSTUFF-1** allows R to set $\mathcal{P}(v)$ as the highest known certificate and speculatively execute transactions of block B_v . 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*).

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 a prepare-certificate for m and has no guarantee that m will commit in the future. Thus, R can erase its local-ledger when it needs to roll back the effects of m 's transactions. Below is the condition for rollback:

Definition 4.7. Rollback Condition. Given two conflicting blocks B_w and B_v such that $w < v$, if a replica R speculatively executed transactions in B_w with prepare-certificate $\mathcal{P}(w)$, R will roll back B_w when R is about to speculatively execute the conflicting B_v with prepare-certificate $\mathcal{P}(v)$ (Lines 25-26).

See Appendix A.2 in [61] for a scenario illustrating rollback in HOTSTUFF-1.

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 of view v fails before broadcasting the prepare-certificate $\mathcal{P}(v)$ for its proposal m to at least $n - f$ 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' speculated in a view w , $w > v$, then it will be rolled back as described in § 3. However, if there are no conflicts, that is, the leader of some view x , $x > v$ observes $\mathcal{P}(v)$ and extends $\mathcal{P}(v)$ in 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 17).

Fortunately, we have an *optimization* that allows replicas to commit and execute B_v at least one phase earlier; if $x = v + 1$ and $\mathcal{P}(v + 1)$ is received, then a replica R can commit B_v , execute transactions, add them to the global-ledger, and reply to their clients (Line 22), which we refer to as the prefix-commit rule.

Recovery Mechanism

A faulty leader 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, a leader can crash before broadcasting the certificate to all replicas.

If the leader $\mathcal{L}_v$ of view v extends its proposal m from the certificate $\mathcal{P}(w)$, $w < v$, then each replica R that receives the proposal needs to validate $\mathcal{P}(w)$ and requires access to the corresponding proposal (say m') of view w . If R does not have access to m' , then it should fetch R from other correct replicas, at least $f + 1$ of which should have it because they voted for m' .

4.2.1 Pacemaker. For a system to make *progress*, at least $n - f$ 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 [31, 71]; group views into *epochs*, each of which contains $f + 1$ consecutive views, and conduct view synchronization at the beginning of every epoch.

The *pacemaker* guarantees that, after *GST*, once the first synchronization is done at view v_s , if a correct replica enters view v , $v \geq v_s$ at time t and sets its timer for view v to expire at time t' , 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 $t' - 2\Delta$, where Δ is the transmission delay bound. Post $t + 2\Delta$, if the leader

L_v for view v waits for an additional message delay, Δ 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 $t + 3\Delta$. For more details, please refer to our extended report [61] and prior works [31, 71].

5 Streamlined Speculation

Basic **HOTSTUFF-1** (§4.1) processes only one proposal every two phases. Like **HOTSTUFF**, we can *streamline the phases* of **HOTSTUFF-1** to ensure that we rotate leaders and inject a new proposal every phase. This has the potential to increase throughput by $2\times$.

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 of each view waits for $n - f$ **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 prepare-certificate it knows and references it in a new proposal with 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 4.6) 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 of 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**.

Prefix Speculation Rule and No-Gap Rule. As in the basic variant, rules guaranteeing safe speculation are needed in streamlined **HOTSTUFF-1** to tackle the Prefix Speculation dilemma described in §3. The enforcement of the Prefix Speculation rule is similar to the basic regime: *a replica R can speculate on a block B_v provided that the prefix of $\mathcal{P}(B_v)$ is committed*. Similarly, the enforcement of the No-Gap rule (Definition 3.2) is necessary, that is, $w = v - 1$. See Appendix A.3 in our extended report [61] for examples of not following the Prefix Speculation rule and the No Gap in streamlined **HOTSTUFF-1**.

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 receives well-formed **NEWVIEW** messages from at least $n - f$ replicas (Figure 3 Line 2), it tries to combine their threshold signature-shares into a threshold signature to create a certificate $\mathcal{P}(v - 1)$ for view $v - 1$ (Line 7). If $\mathcal{L}_v$ fails, it keeps waiting for more **NEWVIEW** messages until it forms a certificate $\mathcal{P}(v - 1)$ or it receives n **NEWVIEW** messages or **PACEMAKER.SHARETIMER**(v) (Line 3). Then, the leader extends its highest certificate to form its new proposal as a **PROPOSE** message m and broadcasts it to all replicas. This proposal includes the view number v , a block B_v of client transactions yet to be proposed, and $\mathcal{P}(v_{lp})$ (Line 5).

Execute and Ledger Update. On receiving a **PROPOSE** message (let's call it m) from the leader (Line 8), R does the following (Lines 9-19):

(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 (Lines 9-10).

(2) If $w = v - 1$ (meets the No-Gap rule), then following the Prefix Speculation Rule: if the predecessor of B_{v-1} is committed, then R *speculatively executes* the transactions in blocks B_{v-1} , adds

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

- 1: **event** Upon `PACEMAKER.ENTREVIEW()` **do**
- 2: Wait until received $n - f$ `NEWVIEW` messages for view v
- 3: Wait until $\mathcal{L}_v$ forms a certificate $\mathcal{P}(v - 1)$ **or** received n `NEWVIEW` messages **or** `PACEMAKER.SHARETIMER(v)`
- 4: Let B_v be a block of client transaction yet to be proposed
- 5: Broadcast $m = \langle \text{PROPOSE}, B_v, v, \mathcal{P}(v_{lp}) \rangle_{\mathcal{L}_v}$
- 6: **event** Received $n - f$ `NEWVIEW` messages with shares for the same proposal of view $v - 1$ **do**
- 7: $\mathcal{P}(v - 1) \leftarrow \text{CREATETHRESHOLDSIGN}(n - f \text{ distinct } \delta_R \text{ shares})$

Backup role (running at each replica R (including leader)):

- 8: **event** Received $\langle \text{PROPOSE}, B_v, \mathcal{P}(w) \rangle_{\mathcal{L}_v}$ **do**
- 9: **if** $\mathcal{P}(w)$ extends $\mathcal{P}(w - 1)$ ***▷commit-rule*** **then**
- 10: Execute all transactions up to (incl.) B_{w-1} , add result to global-ledger and respond to clients
- 11: **if** $w = v - 1$ ***▷No-Gap rule*** **then**
- 12: **if** predecessor of $\mathcal{P}(v - 1)$ is in global-ledger ***▷Prefix Speculation rule*** **then**
- 13: **if** local-ledger state conflicts with B_{v-1} **then**
- 14: Rollback local-ledger to the common ancestor
- 15: Execute all transactions in B_{v-1} speculatively, add result to local-ledger and send client a response ***▷speculatively execute B_{v-1}***
- 16: **if** $w \geq v_{lp}$ **then**
- 17: $\delta_R \leftarrow \text{CREATETHRESHOLD SHARE}(\mathcal{P}(w), v, \text{Hash}(B_v))$
- 18: Send $\langle \text{NEWVIEW}, v + 1, \mathcal{P}(w), \delta_R \rangle_R$ to $\mathcal{L}_{v+1}$
- 19: Call `EXITVIEW()`
- 20: **event** Upon timeout **do**
- 21: Send $\langle \text{NEWVIEW}, v + 1, \mathcal{P}(v_{lp}), \perp \rangle_R$ to $\mathcal{L}_{v+1}$.
- 22: Call `EXITVIEW()`
- 23: **function** `EXITVIEW()` **do**
- 24: $v \leftarrow v + 1$. ***▷disable voting for view v***
- 25: Call `PACEMAKER.COMPLETEDVIEW()`

Fig. 3. Streamlined HOTSTUFF-1.

them to the local-ledger, and sends a reply to respective clients. Before speculatively executing B_{v-1} , R first rolls back its local-ledger if it has speculatively executed a conflicting block (Lines 11-15).

(3) Finally, R checks if w , the view of the certificate $\mathcal{P}(w)$ in m , is not lower than its v_{lp} . If $w \geq v_{lp}$, R updates its highest known certificate $\mathcal{P}(v_{lp})$ with $\mathcal{P}(w)$. Then, R creates a `NEWVIEW` message including $\mathcal{P}(v_{lp})$ and a threshold signature-share δ_R for m , and sends it to the leader of the next view, $\mathcal{L}_{v+1}$ (Lines 16-18).

Timer expiration. In case of timer expiration, the 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 $v + 1$ (Lines 20-22).

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 (Line 25).

Correctness Proof. See Appendix B in our extended report [61] for the correctness proof.

6 Slotting

Rotating leaders in BFT protocols leads to the following challenges:

(1) **Leader-slowness phenomenon.** Rational leaders, who are not malicious but aim to maximize their gains, may delay proposing a block of transactions until as late as possible in their rotation, as they are incentivized to include transactions that yield higher fees. Similarly, block builders participating in a proposer-builder auction may also delay to maximize MEV (maximal extractable value) exploits [34, 89, 91]. If a leader/builder proposes its block too early, it risks filling the block with transactions that offer lower fees than those that may come in the future. Thus, rational leaders and builders may slow down progress, causing increased client latency.

(2) **Tail-forking attack.** In streamlined protocols, the two protocol phases necessary to commit a transaction are spread across the reign of two leaders. The second leader, if malicious, may skip the proposal from the previous leader by pretending that it did not receive enough votes for it, instead of helping drive it to a commit decision⁴.

We address these challenges by introducing *slotting* into the core of streamlined consensus protocols. Slotting enables each leader to propose multiple blocks—one per slot—within its view rotation period. An adaptive slotting mechanism allows leaders to propose as many slots as possible before the view timer expires. Assigning multiple slots per leader/view offers two key benefits: (1) It incentivizes timely proposal of available transactions, as proposing more blocks yields greater rewards; and (2) It eliminates tail-forking attacks for all but the final slot, since a leader can extend its own slot to prevent forking. Additionally, we introduce *carry blocks* in first-slot proposals to protect the last slot of the previous view, if at least $f+1$ correct replicas have voted for it.

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 lower than $B_{i',v'}$. If $v = v'$ and $i < i'$, then block $B_{i,v}$ is ordered lower than $B_{i',v'}$. For instance, in Figure 4, 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, threads to the next view. In the figure, block $B_{2,1}$ includes a certificate for $B_{1,1}$, block $B_{1,2}$ includes a certificate for $B_{4,1}$, and so on.

Second, we introduce a new message type, NEWSLOT, to distinguish a replica's transition to a new slot within the same view from its transition to a new view. Both NEWSLOT and NEWVIEW messages contain threshold signature shares that serve as votes, enabling consensus over slot and view transitions, respectively. To differentiate these votes, replicas sign not only the proposal but also distinct contextual parameters—*New-Slot* and *New-View*. As a result, we define two types of certificates: *New-Slot* and *New-View*. Each *New-View* certificate is further annotated with a parameter f_v , indicating the view in which it was formed, i.e., it is formed by $\mathcal{L}_{f_v}$.

Next, we describe the protocol modifications needed to support slotting. As before, a replica maintains pending, uncommitted blocks of transactions, a local-ledger and the global-ledger. The local state at a replica (refer to Figure 5) includes: (1) $\mathcal{P}(s_{lp}, v_{lp})$, the highest known certificate of view v_{lp} , slot s_{lp} , (2) s, v , the current slot and view, (3) B_h , the highest voted block with hash H_h .

⁴Please refer to Example 6.1 and 6.2 in our extended report for detailed examples of how a rational leader exploits MEV and how a Byzantine leader conducts the tail-forking attack [61].

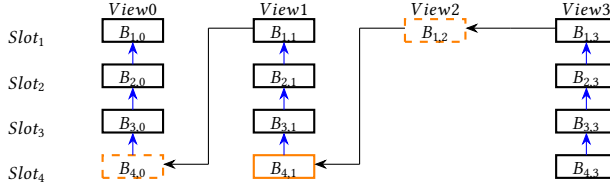


Fig. 4. Chain in HotStuff-1 with Slotting, in which solid black blocks are extended with *New-Slot* certificates. To provide a *self-contained proof* of “no tail-forking” in first-slot proposals, solid orange blocks are extended in way (i), with a *New-View* certificate formed by the next leader; while shaded orange blocks are carried in way (ii), without a certificate.

A well-formed first-slot proposal in view v must provide a *self-contained proof* of “no tail-forking” in one of two ways: (i) **form** and **extend** a *New-View* certificate using votes in $n - f$ *NEWVIEW* messages sent to $\mathcal{L}_v$, e.g., in Figure 4, $B_{1,2}$ extends $\mathcal{P}(4, 1)$; or (ii) **extend** its highest certificate $\mathcal{P}(s_{lp}, v_{lp})$ and **carry** a block B_u .

Definition 6.1. Carry Block: The lowest uncertified block B_u that extends the certificate $\mathcal{P}(s_{lp}, v_{lp})$.

- If $\mathcal{P}(s_{lp}, v_{lp})$ is a *New-View* certificate formed in view fv , then B_u is $B_{1, fv}$. For example, $B_{1,3}$ extends the *New-View* certificate $\mathcal{P}(4, 1)$ with $fv = 2$ and carries the uncertified block $B_{1,2}$.
- If $\mathcal{P}(s_{lp}, v_{lp})$ is a *New-Slot* certificate, then B_u is $B_{s_{lp}+1, v_{lp}}$. For instance, $B_{1,1}$ extends the *New-Slot* certificate $\mathcal{P}(3, 0)$ and carries the uncertified block $B_{4,0}$.

The notions of the *self-contained proof* of “no tail-forking” and *carry block* guarantee that if a correct leader $\mathcal{L}_v$ proposed at least two slots in view v and $f + 1$ correct replicas have voted for its last slot, then view- v slots are protected from *tail-forking attacks*. See further explanations in § 6.2.

Figure 5 and 6 illustrate the pseudocode of streamlined HOTSTUFF-1 with slotting.

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

- (1) well-formed *NEWVIEW* messages for view v , if $s = 1$, or
- (2) well-formed *NEWSLOT* messages for slot $(s - 1, v)$ if $s > 1$.

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 from at least $n - f$ replicas. $\mathcal{L}_v$ delays proposing its first-slot block $B_{1,v}$ until any of the following conditions is met:

- (1) A *New-View* certificate $\mathcal{P}(s_w, w)$, $w < v$, can be formed with $n - f$ *NEWVIEW* messages containing *New-View* threshold signature-shares for the same proposal of slot (s_w, w) .
- (2) $\mathcal{L}_v$ received n *NEWVIEW* messages.
- (3) $\text{PACEMAKER.SHARETIME}(v)$.
- (4) For any slot higher than $\mathcal{P}(s_{lp}, v_{lp})$, $\mathcal{L}_v$ received $n - f$ *NEW-View* messages that do not vote for it.

With these four conditions, it is guaranteed that after *GST*, a correct $\mathcal{L}_v$ can learn the highest certificate across all correct replicas, either by forming it by itself through (1) or learning it from others through (2)-(4). See further explanations in § 6.3.

If condition (1) is satisfied, $\mathcal{L}_v$ proposes its first-slot proposal in way (i): it forms a *New-View* certificate $\mathcal{P}(s_w, w)$ and updates its local highest certificate $\mathcal{P}(s_{lp}, v_{lp})$ to $\mathcal{P}(s_w, w)$. Then, $\mathcal{L}_v$ broadcasts a *PROPOSE* message m containing the block $B_{1,v}$, a batch of new transactions, and the

Local state (replica R) :

- 1: $\mathcal{P}(s_{lp}, v_{lp})$: the highest known certificate formed in view v_{lp} , slot s_{lp}
- 2: s, v : the current slot and view
- 3: B_H : the highest voted block with hash H_H .

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

- 4: **event** Upon PACEMAKER.ENTREVIEW() **do**
- 5: Keep updating $\mathcal{P}(s_{lp}, v_{lp})$ while receiving $n - f$ NEWVIEW messages
- 6: Wait until (1) formed a *NEW-VIEW* certificate $\mathcal{P}(s_w, w)$ **or** (2) received n NEWVIEW messages **or** (3) PACEMAKER.SHARETIMER(v) **or** (4) received $n - k$, $1 \leq k \leq f$, NEWVIEW messages, but there are fewer than $f+1-k$ votes for any slot higher than (s_{lp}, v_{lp})
- 7: **if** $\mathcal{L}_v$ has not proposed $B_{1,v}$ **then**
- 8: Let $B_{1,v}$ be a block of client transactions yet to be proposed
- 9: **if** (1) is satisfied **then**
- 10: Broadcast $m = \langle \text{PROPOSE}, B_{1,v}, 1, v, \mathcal{P}(s_w, w), \perp \rangle_{\mathcal{L}_v}$
- 11: **else**
- 12: $B_u \leftarrow$ the lowest uncertified block that extends $\mathcal{P}(s_{lp}, v_{lp})$
- 13: Broadcast $m = \langle \text{PROPOSE}, B_{1,v}, 1, v, \mathcal{P}(s_{lp}, v_{lp}), H_u \rangle_{\mathcal{L}_v}$
- 14: **event** Received $n - f$ NEWVIEW messages with *NEW-VIEW* signature-shares of $\mathcal{P}(s_w, w)$, $w < v$ **do**
- 15: $\mathcal{P}(s_w, w) \leftarrow \text{CREATENEWVIEWTHRESHOLDSIGN}(n-f \text{ distinct } \delta_H \text{ shares}, f, v = w)$
- 16: **event** Received $n - f$ NEWSLOT messages with *NEW-SLOT* signature-share of $\mathcal{P}(s, v)$ **do**
- 17: $\mathcal{P}(s, v) \leftarrow \text{CREATENEWSLOTTHRESHOLDSIGN}(n - f \text{ distinct } \delta_R \text{ shares})$
- 18: Let $B_{s+1,v}$ be a block of client transaction yet to be proposed
- 19: Broadcast $m = \langle \text{PROPOSE}, B_{s+1,v}, s + 1, v, \mathcal{P}(s, v), \perp \rangle_{\mathcal{L}_v}$
- 20: **event** Received from a *trusted* leader $\mathcal{L}_{v-1}$ a NEWVIEW message with a certificate formed in view $v - 1$ **do**
- 21: Propose $B_{1,v}$ as in Lines 7-13
- 22: **event** Received a REJECT message with $\mathcal{P}(s_{v-1}^*, v - 1)$ **do**
- 23: **if** received from $\mathcal{L}_{v-1}$ a NEWVIEW message with a lower certificate that is formed in view $v - 1$ **then**
- 24: Mark $\mathcal{L}_{v-1}$ as *distrusted*.

Fig. 5. Additional Local State and Leader Role in Streamlined HotStuff-1 with Slotting.

updated certificate $\mathcal{P}(s_{lp}, v_{lp})$. For example, in Figure 4, $\mathcal{L}_1$ proposes $B_{1,1}$ extending the *New-View* certificate $\mathcal{P}(4, 0)$ (see Figure 5, Lines 9–10).

If (2)-(4) is satisfied, $\mathcal{L}_v$ proposes its first-slot proposal in way (ii): $\mathcal{L}_v$ broadcasts a PROPOSE message m that contains $B_{1,v}$, $\mathcal{P}(s_{lp}, v_{lp})$, and H_u , hash of the *lowest uncertified block* B_u that it carries. For example, in Figure 4, $\mathcal{L}_2$ proposes $B_{1,2}$ extending a *New-Slot* certificate $\mathcal{P}(3, 1)$ and carrying $B_{4,1}$ (B_u) (Lines 11-13).

NewSlot: For slot (s, v) , where $s > 1$, the leader $\mathcal{L}_v$ awaits well-formed NEWSLOT messages from at least $n-f$ replicas voting for $B_{s-1,v}$. Once it collects $n-f$ votes, it combines the *New-Slot* signature-shares to create a *New-Slot* certificate $\mathcal{P}(s-1, v)$. After forming $\mathcal{P}(s-1, v)$, $\mathcal{L}_v$ proceeds to propose slot $B_{s,v}$ including $\mathcal{P}(s-1, v)$ (Lines 16-22).

ProposeVote. Upon receiving a proposal $B_{s,v}$, a replica R checks whether any of the following cases are satisfied—based on the slot s , the certificate $\mathcal{P}(s_w, w)$, and the block B_u with hash H_u —before voting (see Figure 6, Lines 3–11). For first-slot proposals, Case 1 provides a valid proof of “no tail-forking” in way (i), while Cases 2 and 3 provide a valid proof in way (ii).

```

1: function SAFESLOT( $s, v, \mathcal{P}(s_w, w), H_u$ ) do
2:   Fetch the carried block  $B_u$  of non-empty hash  $H_u$ 
3:   if  $s = 1$  and  $\mathcal{P}(s_w, w)$  is a NEW-VIEW certificate and  $\mathcal{P}(s_w, w).fv = v$  Case 1 then
4:     return true
5:   else if  $s = 1$  and  $\mathcal{P}(s_w, w)$  is a NEW-VIEW certificate and  $\mathcal{P}(s_w, w).fv < v$  and  $B_u.slot = 1$  and  $B_u.view = \mathcal{P}(s_w, w).fv$  Case 2 then
6:     return true
7:   else if  $s = 1$  and  $\mathcal{P}(s_w, w)$  is a NEW-SLOT certificate and  $B_u.slot = s_w + 1$  and  $B_u.view = w$  Case 3 then
8:     return true
9:   else if  $s > 1$  and  $\mathcal{P}(s_w, w)$  is a NEW-SLOT certificate and  $s_w = s - 1$  and  $w = v$  Case 4 then
10:    return true
11:  return false

Backup role (running at each replica  $R$  (including leader)) :
12: event Received  $\langle \text{PROPOSE}, B_{s,v}, \mathcal{P}(s_w, w), H_u \rangle_{\mathcal{L}_v}$  do
13:   if  $\mathcal{P}(s_w, w)$  extends  $\mathcal{P}(s_w - 1, w)$  commit-rule-case1 then
14:     Execute all transactions up to (incl.)  $B_{s_w-1,w}$ , add result to global-ledger and respond to clients
15:   else if  $s_w = 1$  and  $\mathcal{P}(s_w, w)$  extends  $\mathcal{P}(s_w-1, w-1)$  commit-rule-case2 then
16:     Execute all transactions up to (incl.)  $B_{s_w-1,w-1}$ , add result to global-ledger and respond to clients

17:   if ( $s = s_w + 1$  and  $v = w$ ) or ( $s = 1$  and  $v = w + 1$ ) No-Gap rule
       and predecessor of  $\mathcal{P}(s_w, w)$  is in global-ledger Prefix Speculation rule then
18:     if local-ledger state conflicts with  $B_{s_w,w}$  then
19:       Rollback local-ledger to the common ancestor
20:     Execute all transactions in  $B_{s_w,w}$  speculatively, add the result to local-ledger and send the client a response

21:   if SAFESLOT( $s, v, \mathcal{P}(s_w, w), H_u$ ) then
22:      $\delta_R \leftarrow \text{CREATETHRESHOLD SHARE}(\mathcal{P}(s_{lp}, v_{lp}), \text{Hash}(B_{s,v}), H_u, \text{New-Slot})$ 
23:     Send  $\langle \text{NEWSLOT}, s, v, \mathcal{P}(s_{lp}, v_{lp}), \delta_R \rangle_R$  to  $\mathcal{L}_v$ 
24:   else
25:     Send  $\langle \text{REJECT}, s, v, \mathcal{P}(s_{lp}, v_{lp}) \rangle_R$  to  $\mathcal{L}_v$ 

26:    $s \leftarrow s + 1$  disable voting for slot s

27: event Upon timeout do
28:    $\delta_h \leftarrow \text{CREATETHRESHOLD SHARE}(\mathcal{P}(s_{lp}, v_{lp}), H_h, \text{New-View})$ 
29:   Send  $\langle \text{NEWVIEW}, v + 1, \mathcal{P}(s_{lp}, v_{lp}), H_h, \delta_h \rangle_R$  to  $\mathcal{L}_{v+1}$ 
30:    $v \leftarrow v + 1, s \leftarrow 1$  disable voting for view v
31:   Call PACEMAKER.COMPLETEDVIEW()

```

Fig. 6. Backup Role in Streamlined HOTSTUFF-1 + Slotting.

Case 1: $s = 1$; $B_{1,v}$ extends a *New-View* certificate $\mathcal{P}(s_w, w)$ such that $\mathcal{P}(s_w, w).fv = v$.

Case 2: $s = 1$; $B_{1,v}$ extends a *New-View* certificate $\mathcal{P}(s_w, w)$ such that $\mathcal{P}(s_w, w).fv < v$; and $B_{1,v}$ carries B_u such that $B_u.slot = 1$ and $B_u.view = \mathcal{P}(s_w, w).fv$.

Case 3: $s = 1$; $B_{1,v}$ extends a *New-Slot* certificate $\mathcal{P}(s_w, w)$ and carries $B_{s_w+1,w}$.

Case 4: $s > 1$; $B_{s,v}$ extends a *New-Slot* certificate $\mathcal{P}(s_w, w)$ such that $s_w = s - 1$ and $w = v$.

Then, R checks if its highest certificate $\mathcal{P}(s_{lp}, v_{lp})$ is lexicographically not greater than $\mathcal{P}(s_w, w)$. If so, R sends a NEWSLOT message containing a *New-Slot* signature-share of $B_{s,v}$ (Line 22).

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 $\mathcal{P}(s_{lp}, v_{lp})$, its highest certificate, H_h , hash of its highest voted block, and δ_h , a *New-View* signature share for $\mathcal{P}(s_{lp}, v_{lp})$ and H_h (Lines 27-31).

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 4.6) to add a transaction to the global-ledger.

However, as we form a two-dimensional 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 13) (2) $s_w = 1$: commits block $B_{s_w-1, w-1}$ if $\mathcal{P}(s_w, w)$ extends $\mathcal{P}(s_w - 1, w - 1)$ (Line 15).

Of special note are the *uncertified carry blocks* in the first-slot blocks, that are viewed as a part of the first-slot blocks. If a first-slot block $B_{1,v}$ contains H_u of a carry block B_u , then B_u gets committed only when $B_{1,v}$ is committed.

Speculation. Replicas may speculate on a block $B_{s_w, w}$ when it satisfies the Prefix Speculation Rule and No-Gap 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 the prefix of $B_{s_w, w}$ is committed and $B_{s_w, w}$ is from the immediately preceding slot (Line 17), i.e., (1) $s = s_w + 1, v = w$; or (2) $s = 1, v = w + 1$.

6.2 Tolerance to Tail-Forking

Now we show how the HOTSTUFF-1 with slotting mitigates tail-forking attacks. We denote by $B_{s-1, v}, B_{s, v}$, the last two slots of view v with a correct leader $\mathcal{L}_v$, where $B_{s, v}$ extends $B_{s-1, v}$. It is guaranteed that if $\mathcal{L}_v$ proposed at least two slots and at least $f + 1$ correct replicas have voted for its last slot $B_{s, v}$, then $B_{s, v}$ and all preceding slots in view v are protected from *tail-forking attacks*.

That is, if at least $f+1$ correct replicas have voted for $B_{s, v}$, it becomes impossible to form a *New-View* certificate $\mathcal{P}(s-1, v)$, as those $f+1$ correct replicas will vote for $B_{s, v}$ rather than $B_{s-1, v}$ in their **NEWVIEW** messages. Consequently, $B_{1, v+1}$ can extend either a *New-Slot* certificate $\mathcal{P}(s-1, v)$ or a *New-View* certificate $\mathcal{P}(s, v)$. If it extends the *New-Slot* $\mathcal{P}(s-1, v)$, then by Case 3 of **PROPOSEVOTE** phase, it must *carry* $B_{s, v}$; otherwise, it extends the *New-View* certificate $\mathcal{P}(s, v)$. In either case, $B_{s, v}$ and all preceding slots in view v are not *tail-forked*.

6.3 Advancing at Network Speed with Trusted Previous Leaders

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 the pre-determined maximal network delay Δ , unless there is a fault.

STREAMLINED HOTSTUFF-1 with *slotting* introduces a new challenge: $\mathcal{L}_v$ does not know in advance the highest slot s proposed in view $v-1$, since each leader attempts to propose as many slots as possible before its view expires, and the number of slots per view is adaptive. If a correct leader $\mathcal{L}_{v-1}$ fails to broadcast its final slot to at least $n-f$ well-behaving replicas before their view timers expire, the next leader $\mathcal{L}_v$ may be unable to form a *New-View* certificate and must wait for an $O(\Delta)$ delay to receive **NEWVIEW** messages from all correct replicas.

To avoid this unintended $O(\Delta)$ delay between two correct leaders, we introduce the notion of *trusted* and *distrusted* previous leaders. Initially, each leader trusts its previous leader in the rotation. Upon receiving a **NEWVIEW** message from a *trusted* previous leader $\mathcal{L}_{v-1}$ that includes a certificate formed in view $v-1$ (Figure 5, Line 20), i.e., a *New-Slot* certificate of view $v-1$ or a *New-View* certificate with $fv = v-1$, $\mathcal{L}_v$ immediately proposes its first-slot extending $\mathcal{P}(s, v-1)$. This is safe

because no correct replica can hold a higher certificate than that of a correct $\mathcal{L}_{v-1}$ when exiting view $v-1$, assuming a certificate was formed in view $v-1$. The trusted/distrusted mechanism thus enables $\mathcal{L}_v$ to propose its first-slot block at network speed, avoiding unnecessary delays.

However, a Byzantine previous leader $\mathcal{L}_{v-1}$, initially *trusted* by $\mathcal{L}_v$, may conceal the highest certificate it has formed and sent to other correct replicas. This can cause $\mathcal{L}_v$'s first-slot proposal to be rejected by a correct replica R that has already received the higher certificate. Upon rejection, R sends a REJECT message to $\mathcal{L}_v$ containing its highest certificate (Figure 6, Line 25). If $\mathcal{L}_v$ had previously received a NEWVIEW message from $\mathcal{L}_{v-1}$ containing a lower certificate formed in view $v-1$ (Figure 5, Line 23), it then marks $\mathcal{L}_{v-1}$ as *distrusted*. In future views where $\mathcal{L}_v$ becomes leader again, it no longer trusts $\mathcal{L}_{v-1}$ and follows the four conditions described in §6.1 when entering the view. As a result, each malicious leader $\mathcal{L}_{v-1}$ can conceal its highest certificate at most once after GST, without compromising liveness.

If the previous leader is *distrusted*, the four conditions for proposing the first slot ensure that, after GST, a correct leader $\mathcal{L}_v$ can learn the highest certificate known to any correct replica. If condition (1) holds, then at least $f+1$ correct replicas did not vote for any slot higher than the formed certificate $\mathcal{P}(s_w, w)$, implying that no higher certificate could exist. If condition (2) or (3) holds, the Pacemaker guarantees that $\mathcal{L}_v$ receives NEWVIEW messages from all correct replicas, thereby acquiring the highest certificate. Under condition (4), the highest votes contained in the NEWVIEW messages reveal that no higher certificate could have been formed.

7 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 f additional required responses on HOTSTUFF-1.
- (3) Impact of leader-slowness, tail-forking, and rollbacks.

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 specific sizes when necessary.

Implementation. We implement all the protocols in APACHE RESILIENTDB (incubating) [12]; 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 $n - f$ digital signatures (from distinct replicas) as a certificate.

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

- (1) HOTSTUFF. First streamlined BFT consensus protocol; requires 7 half-phases to reach consensus on a client transaction (total 9 half-phases including client request and response).
- (2) HOTSTUFF-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. Streamlined BFT consensus protocol with speculative execution that requires 3 half-phases for speculative response (total 5 half-phases).
- (2) HOTSTUFF-1 (**with Slotting**).

Workloads. We use two workloads: YCSB [38] and TPC-C [1]:

- (1) YCSB. Key-value store write operations that access a database of 600k records.

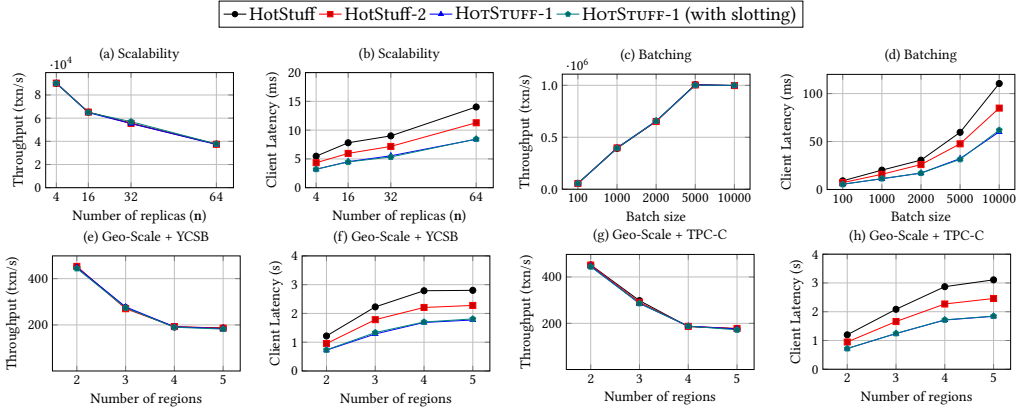


Fig. 7. Scalability Plots.

(2) TPC-C. Online transaction processing (OLTP) operations that access a database of 260k records, simulating a complex warehouse and order management environment. Unless explicitly stated, we use YCSB as the default workload.

Metrics. We focus on two metrics:

- (1) *Throughput* – the maximum number of transactions per second for which the system completes consensus.
- (2) *Client Latency* – the average duration between the time a client sends a transaction to the time the client receives a matching quorum of responses ($f+1$ for HOTSTUFF/HOTSTUFF-2 and $n-f$ for HOTSTUFF-1) for that transaction.

7.1 Scalability

Impact of the number of replicas In Figures 7 (a) and (b), we present various system metrics as a function of the number of replicas; we increase the number of replicas from $n = 4$ to $n = 64$.

As expected, an increase in the number of replicas causes a proportional decrease in the throughput for all the protocols due to an $O(n)$ increased message complexity, which decreases available bandwidth and increases the computational work at each replica. HOTSTUFF-1, with or without slotting, yields the same throughput as HOTSTUFF/HOTSTUFF-2 because the message complexity remains the same for all the streamlined protocols.

An increase in the number of replicas also causes a proportional increase in the client latency for all the protocols due to an $O(n)$ increased message complexity, which increases the time duration for a leader to collect a quorum of threshold shares and to form a certificate. Moreover, each client needs to wait longer for a larger quorum of messages to arrive. This implies that HOTSTUFF-1 clients should incur higher latency as they must wait for f more responses. However, HOTSTUFF-1 yields lower latency because speculation guarantees an early finality confirmation. HOTSTUFF-1, with or without slotting, yields 41.5% and 24.2% (for small setups) and 38.5% and 22.7% (for large setups) less client latency in comparison to HOTSTUFF and HOTSTUFF-2.

Impact of Batch Size Next, in Figures 7 (c) and (d), we increase the number of transactions per batch (batch size) from 100 to 10000 and run consensus among $n = 32$ replicas.

For all protocols, increasing the batch size improves throughput until either bandwidth or compute resources are saturated, beyond which throughput tapers off. The throughput gain at smaller batch sizes is due to reduced consensus overhead and fewer messages being processed. At larger batch sizes (around 5000), all protocols become compute-bound before reaching bandwidth

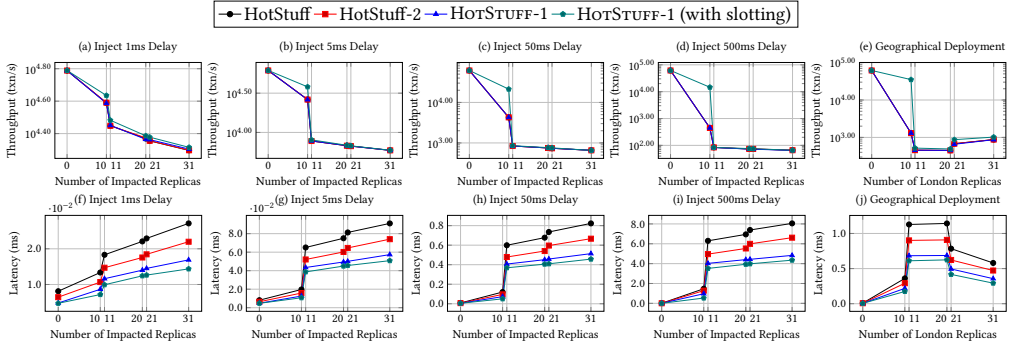


Fig. 8. Performance with Varying Network Conditions.

saturation, as the benefits of reduced consensus overhead are offset by the increased cost of proposing (for leaders) and processing (for replicas) larger batches. In contrast, client latency increases with batch size, as proposing and processing larger batches takes more time in each view.

Geo-Scale Scalability In Figures 7 (e–h), we deploy replicas across the globe, varying the number of geographical regions from 2 to 5—North Virginia, Hong Kong, London, São Paulo, and Zurich—and uniformly distribute $n = 32$ replicas across these regions. These experiments use both the YCSB and TPC-C benchmarks. We observe that all protocols exhibit similar trends across both benchmarks, as high inter-regional round-trip times limit throughput and increase latency.

As the number of regions increases from 2 to 5, all protocols experience up to a 59% drop in throughput and a 159.4% increase in latency. Nevertheless, the general trend remains consistent: HOTSTUFF-1 matches the throughput of other protocols while achieving the lowest latency.

7.2 Impact of the f Additional Responses

We now experimentally validate our claim: although HOTSTUFF-1 clients wait for f additional responses compared to HOTSTUFF/HOTSTUFF-2 clients, HOTSTUFF-1 always yields the lowest latency for clients.

Injecting Message Delay. We begin by evaluating the impact of delayed messages on client latency. This experiment demonstrates that even when more than $f + 1$ replicas experience high message delays, HOTSTUFF-1 clients do not incur increased latencies. The setup is as follows: (1) We deploy $n = 31$ replicas. (2) Based on prior experiments, the client latencies for HOTSTUFF-1/HOTSTUFF-2/HOTSTUFF are approximately 5 ms/7 ms/9 ms, respectively. We inject increasing message delays $\delta \in \{1 \text{ ms}, 5 \text{ ms}, 50 \text{ ms}, 500 \text{ ms}\}$. (3) We vary the number of impacted replicas $k \in \{0, f, f+1, n-f-1, n-f, n\}$, i.e., $k = 0, 10, 11, 20, 21, 31$. Figures 8 (a–d) and (f–i) present the results of these experiments.

For all protocols, as the number of impacted replicas increases, latency increases and throughput decreases due to the delayed message transmission to and from these replicas. The impact is most pronounced when increasing from $k = f$ (10) to $k = f+1$ (11), as every certificate formed by the leader must now include at least one signature share from an impacted replica (since certificates require $n-f$ signatures). These results further support our claim that the primary bottleneck in these protocols lies in achieving consensus, rather than in responding to clients.

As the number of impacted replicas increases from $k = n-f-1$ (20) to $k = n-f$ (21), client latencies in HOTSTUFF and HOTSTUFF-2 increase sharply, whereas HOTSTUFF-1 shows only a moderate increase. This is because, when $k \geq n-f$, clients can receive at most f responses from non-impacted replicas, causing latency to be dominated by the slower, impacted replicas.

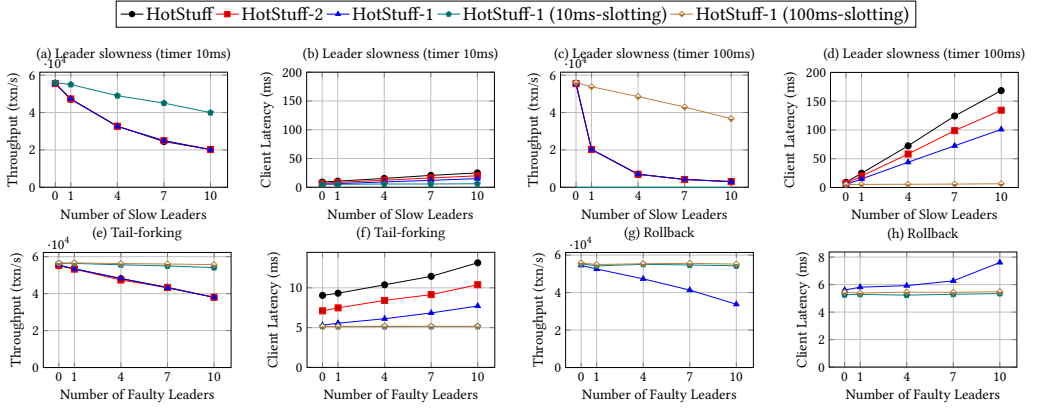


Fig. 9. Impact of varying the number of faulty replicas (leader slowness, tail-forking, and rollback).

When $k \leq f$, HOTSTUFF-1 with *slotting* yields better performance than all other protocols because slotting allows the non-impacted replicas to propose more blocks during their views.

Geographical Deployment. Next, we deploy $n = 31$ replicas across two geographically distant regions: North Virginia and London, with all clients located in North Virginia. We vary the number of replicas placed in London, denoted by $k \in \{0, f, f+1, n-f-1, n-f, n\}$. The results, shown in Figures 8(k) and (l), illustrate the impact of increasing geographic separation among replicas.

When $k \leq f$ (10) or $k \geq n-f$ (21), leaders in North Virginia and London, respectively, can form certificates using votes from $n-f$ replicas within their own region. In contrast, when k is between $f+1$ (11) and $n-f-1$ (20), forming a certificate requires at least one vote from the remote region, leading to degraded throughput and latency. Performance is better when $k \leq f$ than when $k \geq n-f$ because most leaders are co-located with clients in North Virginia. When $k \leq f$ or $k \geq n-f$, HOTSTUFF-1 with *slotting* outperforms other protocols, as slotting enables leaders with $n-f$ co-located replicas to propose more blocks per view.

7.3 Failure Resiliency

Leader slowness phenomenon.

We now study the impact of leader slowness (§6) on streamlined protocols by varying the number of slow leaders from 0 to f , with $n = 32$ replicas, a batch size of 100, and two timeout settings: 10 ms and 100 ms. A slow leader does not propose until the end of its view duration. Figures 9(a)–(d) present the results.

Slow leaders degrade throughput and client latency in all protocols except HOTSTUFF-1 with *slotting*. In HOTSTUFF-1, each leader can propose multiple slots, eliminating delays associated with leader slowness. Moreover, the larger the timeout period, the more batches a leader can propose, further improving performance. For example, with a timeout of 10 ms, HOTSTUFF-1 (with slotting) experiences only a 1.8% and 28.7% drop in throughput and a 0.9% and 18.5% increase in latency with 1 and $f = 10$ slow leaders, respectively. In contrast, other protocols suffer 14.5% and 63.5% lower throughput and 18.7% and $2.8\times$ higher latency under the same conditions. Similarly, with a timeout of 100 ms, HOTSTUFF-1 incurs 3.9% and 34.4% lower throughput and 5.7% and 27.1% higher latency with 1 and $f = 10$ slow leaders, while other protocols see throughput drop by 63.4% and 94.5%, and latency increase by $2.81\times$ and $19\times$, respectively.

Tail-forking attack. Similar to the leader slowness phenomenon, the tail-forking attack seeks to increase system latency by preventing proposals from correct leaders from being committed (§6).

In this experiment, we vary the number of faulty leaders from 0 to f , using $n = 32$ replicas and a batch size of 100. As shown in Figures 9(e) and (f), a faulty leader in view v ignores the certificate from the proposal in view $v-1$ and instead extends its proposal from the certificate of view $v-2$.

As before, faulty leaders degrade the performance of all protocols except HOTSTUFF-1 with *slotting*. In HOTSTUFF-1 with *slotting*, each leader can propose multiple batches, and a faulty leader can at most suppress the final slot, mitigating the impact of the attack. HOTSTUFF-1 with *slotting* demonstrates greater resilience, particularly with longer timeout periods. For instance, with $f = 10$ faulty leaders, HOTSTUFF-1 (with *slotting*) shows only a 4.1% and 1.4% reduction in throughput under timeout settings of 10 ms and 100 ms, respectively—compared to the no-failure case. In contrast, other protocols suffer a 31.6% drop in throughput under the same conditions. Similarly, HOTSTUFF-1 experiences minimal change in latency, while other protocols exhibit up to a 45.3% increase in client latency with $f = 10$ faulty leaders.

Rollback. In HOTSTUFF-1, speculation on uncommitted transactions may require replicas to roll back speculated transactions. In Figures 9(g)–(h), we vary the number of faulty leaders from 0 to f and allow each to force up to f correct replicas to roll back transactions, using $n = 32$ replicas and a batch size of 100. Notably, in HOTSTUFF-1 with *slotting*, a faulty leader $\mathcal{L}_v$ can only force rollbacks of the last slot in the preceding view $v-1$; it cannot skip any slot in its own view. Faulty leaders degrade throughput and latency in HOTSTUFF-1 without *slotting*. With $f = 10$, HOTSTUFF-1 without *slotting* suffers a 38.1% drop in throughput and a 35.8% increase in latency relative to the no-failure case. In contrast, rollback attacks have minimal impact on HOTSTUFF-1 with *slotting*.

8 Related Work

Extensive literature exists on consensus, with numerous studies (e.g., [8, 9, 11, 13, 16, 17, 20, 27, 37, 45, 53, 67, 74, 75, 83, 87, 92, 93, 97, 111]) focused on enhancing consensus systems [21, 22, 47, 49, 52, 54, 55, 57, 60, 66, 68, 77, 78, 81, 90, 96, 99, 110, 112].

Speculation. Protocols belonging to the PBFT family [4, 45, 67] have explored an *optimistic fast-path* approach to speculation. 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 [48, 56]. 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 communication costs to linear, enabling regular leader replacement at no additional communication cost or drop in system throughput. HOTSTUFF-2 [79] 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 for two-phase streamlined and linear latency. However, Fast-HotStuff [59] and Jolteon [41] have quadratic complexity in view-change; AAR [5] employs expensive zero-knowledge proofs; Wendy [42] relies on a new aggregate signature construction (and is super-linear); Marlin [103] introduces an additional *virtual block*, offering leaders one more chance to propose a block extending the highest certificate that is supported by all correct replicas.

Parallel Dissemination. Slotting is complementary to the prior multi-leader protocols like RCC [46, 51], MirBFT [102], and SpotLess [62]. These protocols focus mostly on increasing *throughput*, and a majority of them have a HOTSTUFF-core. Thus, their designs are orthogonal to this paper. Any reduction in latency, the elimination of leader slowness phenomena, and tail-forking attacks will improve them. Autobahn [43] presents a data dissemination protocol that separates the task of disseminating client requests from consensus. It allows all replicas, in parallel, to batch and broadcast client requests. However, after dissemination, Autobahn employs PBFT to reach consensus on the execution order for all requests. Thus, Autobahn is orthogonal to the design of

HOTSTUFF-1; the PBFT consensus in Autobahn can be replaced with HOTSTUFF-1 to yield lower latency. DAG-based consensus protocols [30, 35, 64, 65, 80, 100, 101, 108] decouple data dissemination from consensus by leveraging *reliable broadcast (RBC)* mechanisms [24]. These protocols construct a *Directed Acyclic Graph (DAG)* of blocks generated by distinct replicas, enabling high throughput. However, this comes at the cost of increased latency introduced by RBC. Recent works [14, 18, 98] have focused on reducing the latency of DAG-based consensus protocols. In this context, we posit that speculative execution offers a promising approach to further reduce latency.

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 [76, 85, 86, 107] have $O(n^3)$ worst-case message complexity. RareSync[31] and Lewis-Pye [71] reduce the worst-case message complexity to $O(n^3)$ but face $O(n\Delta)$ latency in the presence of faulty leaders. Fever [72] removes the $O(n\Delta)$ latency but assumes a synchronous start of replicas. Lumiere [73] eliminates the need for the assumption and maintains all other properties of Fever. SpotLess [62] adopts a rapid view synchronization mechanism similar to FastSync [107], but embeds view synchronization into the BFT consensus workflow, eliminating the need for a separate sub-protocol.

Leader Slowness. The leader-slowness attack is a well-known problem in blockchains [34, 89, 91]. Prior work has illustrated that in Ethereum, for 59% of blocks, proposers have earned higher MEV rewards than block rewards [89], and any additional delay in proposing can help maximize their MEVs [95]. 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 [15]. (ii) Assign block rewards proportional to the number of attestations; a delayed block will receive fewer attestations and thus reduced block rewards [94]. However, if MEV rewards exceed total block rewards, the proposer makes a profit despite losing any block reward.

Tail-forking attack. BeeGees [44] describes the problem of tail-forking. They present an elegant solution to this problem by requiring an incoming leader to collect $n - f$ votes for the previous tail or prove that it cannot collect them. Unfortunately, proving is implemented with either quadratic communication complexity or by generating computationally-prohibitive SNARKs.

Real-World Deployments. Several deployed blockchain systems, such as Espresso Systems HotShot [19], Flow Networks [40], Meter [82] have expressed a latency-over-everything emphasis. Early adopters of HOTSTUFF, DiemBFT [36], and Aptos that uses a two-phase variant of DiemBFT, Ditto [41], demonstrate the importance of latency. Recently, Spacecoin [3] unveiled plans to launch a trust platform operating within satellite-cubes in orbit, where latency is paramount because the link from Earth to satellites is slow. All of these systems may benefit from incorporating HOTSTUFF-1.

9 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.

Acknowledgments

This work is partially funded by NSF Award Number 2245373.

References

- [1] 2010. TPC-C Benchmark: Standard Specification. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf. Accessed: 2025-01-16.

- [2] 2020. Principles for Financial Market Infrastructures (PFMI). https://www.bis.org/cpmi/info_pfmi.htm.
- [3] 2024. Spacecoin Blue Paper. <https://github.com/spacecoinxyz/research/blob/main/publications/Blue-Paper-Spacecoinxyz.pdf>.
- [4] 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>
- [5] Mark Abspoel, Thomas Attema, and Matthieu Rambaud. 2020. Malicious security comes for free in consensus with leaders. *Cryptology ePrint Archive* (2020).
- [6] 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. doi:10.1109/TDSC.2010.70
- [7] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: A Cross-application Permissioned Blockchain. *Proc. VLDB Endow.* 12, 11 (2019), 1385–1398. doi:10.14778/3342263.3342275
- [8] 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. doi:10.1145/3448016.3452807
- [9] 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.
- [10] 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 13th EuroSys Conference*. ACM, 30:1–30:15. doi:10.1145/3190508.3190538
- [11] 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. doi:10.1109/ICDCS51616.2021.00045
- [12] Apache Software Foundation. 2023. Apache ResilientDB (Incubating). <https://resilientdb.incubator.apache.org>
- [13] 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. doi:10.1109/SCC49832.2020.00062
- [14] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2024. Shoa!+: High throughput dag bft can be fast! *arXiv preprint arXiv:2405.20488* (2024).
- [15] Aditya Asgaonkar. 2021. Proposer LMD Score Boosting, Ethereum Consensus-Specs. <https://github.com/ethereum/consensus-specs/pull/2730>
- [16] 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. doi:10.1145/2658994
- [17] 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. doi:10.1109/ICDCS.2013.53
- [18] 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).
- [19] Jeb Bearer, Benedikt Bünz, Philippe Camacho, Binyi Chen, Ellie Davidson, Ben Fisch, Brendon Fish, Gus Gutoski, Fernando Krell, Chengyu Lin, et al. 2024. The espresso sequencing network: Hotshot consensus, tiramisu data-availability, and builder-exchange. *Cryptology ePrint Archive* (2024).
- [20] 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. doi:10.1145/3284764.3284767
- [21] 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. doi:10.1145/3590140.3592848
- [22] 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. doi:10.1145/3320269.3384752
- [23] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptography and information security*. Springer, 514–532.
- [24] Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)* 32, 4 (1985), 824–840.
- [25] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *CoRR* abs/1807.04938 (2018).

- [26] Vitalik Buterin. 2013. Ethereum White Paper: A Next-Generation Smart Contract and Decentralized Application Platform. <https://ethereum.org/en/whitepaper/>.
- [27] 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. doi:10.4230/LIPIcs.DISC.2017.1
- [28] Miguel Castro. 2001. *Practical Byzantine Fault Tolerance*. Ph. D. Dissertation. Massachusetts Institute of Technology. <https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/thesis-mcastro.pdf>
- [29] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461. doi:10.1145/571637.571640
- [30] Junchao Chen, Alberto Sonnino, Lefteris Kokoris-Kogias, and Mohammad Sadoghi. 2024. Thunderbolt: Causal Concurrent Consensus and Execution. *arXiv preprint arXiv:2407.09409* (2024).
- [31] Pierre Civi, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. 2022. Byzantine Consensus Is $\Theta(n^2)$: 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. doi:10.4230/LIPIcs.DISC.2022.14
- [32] 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.
- [33] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.
- [34] 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:121212213>
- [35] 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. doi:10.1145/3492321.3519594
- [36] Diem. 2020. DiemBFT consensus protocol. <https://github.com/diem/diem/tree/latest/consensus>
- [37] 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. doi:10.1109/TKDE.2017.2781227
- [38] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1085–1100. doi:10.1145/3035918.3064033
- [39] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323. doi:10.1145/42282.42283
- [40] Flow. 2025. Flow: The Blockchain for Open Worlds. <https://flow.com>. Accessed: 2025-01-21.
- [41] 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.
- [42] Neil Girdharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. 2021. No-Commit Proofs: Defeating LiveLock in BFT. <https://eprint.iacr.org/2021/1308>
- [43] Neil Girdharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. 2024. Autobahn: Seamless high speed BFT. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 1–23.
- [44] Neil Girdharan, 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. doi:10.1145/3583668.3594572
- [45] 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. doi:10.1109/DSN.2019.00063
- [46] 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>
- [47] Suyash Gupta, Mohammad Javad Amiri, and Mohammad Sadoghi. 2023. Chemistry behind Agreement. In *13th Conference on Innovative Data Systems Research, CIDR*. www.cidrdb.org. <https://www.cidrdb.org/cidr2023/papers/p85-gupta.pdf>

- [48] Suyash Gupta, Jelle Hellings, Sajjad Rahnema, 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. doi:10.5441/002/edbt.2021.27
- [49] 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. doi:10.4230/LIPIcs.DISC.2019.44
- [50] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. *Fault-Tolerant Distributed Transactions on Blockchain*. Morgan & Claypool. doi:10.2200/S01068ED1V01Y202012DTM065
- [51] 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. doi:10.1109/ICDE51399.2021.00124
- [52] Suyash Gupta, Sajjad Rahnema, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (2020), 868–883. doi:10.14778/3380750.3380757
- [53] Suyash Gupta, Sajjad Rahnema, 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. doi:10.1109/ICDE55515.2023.00030
- [54] Suyash Gupta, Sajjad Rahnema, 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. doi:10.1145/3552326.3587455
- [55] Suyash Gupta, Sajjad Rahnema, 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. doi:10.1109/ICDCS47774.2020.00012
- [56] Jelle Hellings, Suyash Gupta, Sajjad Rahnema, and Mohammad Sadoghi. 2022. On the Correctness of Speculative Consensus. arXiv:2204.03552 [cs.DB] <https://arxiv.org/abs/2204.03552>
- [57] 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>
- [58] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
- [59] 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).
- [60] Dakai Kang, Junchao Chen, Tien Tuan Anh Dinh, and Mohammad Sadoghi. 2025. FairDAG: Consensus Fairness over Concurrent Causal Design. *arXiv preprint arXiv:2504.02194* (2025).
- [61] Dakai Kang, Suyash Gupta, Dahlia Malkhi, and Mohammad Sadoghi. 2024. HotStuff-1: Linear Consensus with One-Phase Speculation. *CoRR* abs/2408.04728 (2024). doi:10.48550/ARXIV.2408.04728 arXiv:2408.04728
- [62] Dakai Kang, Sajjad Rahnema, 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.
- [63] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography* (2nd ed.). Chapman and Hall/CRC.
- [64] 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.
- [65] 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, Vol. 281)*, Rotem Oshman (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:22.
- [66] 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.
- [67] 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. doi:10.1145/1658357.1658358
- [68] 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.
- [69] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (2001), 51–58. doi:10.1145/568425.568433 Distributed Computing Column 5.

- [70] Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. 2019. FairLedger: A Fair Blockchain Protocol for Financial Institutions. In *International Conference on Principles of Distributed Systems*. <https://api.semanticscholar.org/CorpusID:182952373>
- [71] Andrew Lewis-Pye. 2022. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model. <https://arxiv.org/abs/2201.01107>
- [72] Andrew Lewis-Pye and Ittai Abraham. 2023. Fever: optimal responsive view synchronisation. *arXiv preprint arXiv:2301.09881* (2023).
- [73] 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.
- [74] 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.
- [75] 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>
- [76] 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.
- [77] Hanzheng Lyu, Shaokang Xie, Jianyu Niu, Ivan Beschastnikh, Yinqian Zhang, Mohammad Sadoghi, and Chen Feng. 2024. Orthrus: Accelerating Multi-BFT Consensus through Concurrent Partial Ordering of Transactions. *arXiv preprint arXiv:2501.14732* (2024).
- [78] 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. doi:10.1109/EDCC.2019.00022
- [79] Dahlia Malkhi and Kartik Nayak. 2023. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive* (2023).
- [80] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2023. BBKA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG. *CoRR* abs/2310.06335 (2023).
- [81] Tejas Mane, Xiao Li, Mohammad Sadoghi, and Mohsen Lesani. 2024. AVA: Fault-tolerant Reconfigurable Geo-Replication on Heterogeneous Clusters. *arXiv preprint arXiv:2412.01999* (2024).
- [82] Meter.io. 2025. Meter: Decentralized Finance Infrastructure. <https://meter.io>. Accessed: 2025-01-21.
- [83] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 31–42.
- [84] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. <https://bitcoin.org/bitcoin.pdf>
- [85] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. 2021. Cogsworth: Byzantine view synchronization. (2021).
- [86] Oded Naor and Idit Keidar. 2024. Expected linear round synchronization: The missing link for linear byzantine smr. *Distributed Computing* 37, 1 (2024), 19–33.
- [87] Faisal Nawab and Mohammad Sadoghi. 2023. Consensus in Data Management: From Distributed Commit to Blockchain. *Found. Trends Databases* 12, 4 (2023), 221–364. doi:10.1561/19000000075
- [88] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.
- [89] Burak Öz, Benjamin Kraner, Nicolò Vallerano, 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. doi:10.1145/3605768.3623563
- [90] Sajjad Rahnema, 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.
- [91] Ethereum Roadmap. 2024. Proposer-Builder Separation. <https://ethereum.org/en/roadmap/pbs/>
- [92] Christian Rondonini, 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. doi:10.1109/SCC49832.2020.00056
- [93] 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. doi:10.1007/s00778-020-00646-1

- [94] Caspar Schwarz-Schilling. 2022. Retroactive Proposer Rewards. <https://notes.ethereum.org/@casparschwa/S1vcyXZL9>
- [95] 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 (LIPIcs), Vol. 282)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:17. doi:10.4230/LIPIcs.AFT.2023.30
- [96] 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. doi:10.1145/3373376.3378469
- [97] 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. doi:10.1145/3460120.3484566
- [98] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2024. Sailfish: Towards Improving the Latency of DAG-based BFT. *Cryptology ePrint Archive*, Paper 2024/472.
- [99] 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. doi:10.1145/3465480.3466922
- [100] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. 2023. Shoal: Improving DAG-BFT latency and robustness. *arXiv preprint arXiv:2306.03058* (2023).
- [101] 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.
- [102] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. <http://arxiv.org/abs/1906.05552>
- [103] Xiao Sui, Sisi Duan, and Haibin Zhang. 2022. Marlin: Two-Phase BFT with Linearity. In *2022 52nd Annual IEEE/IFIP Int'l Conference on Dependable Systems and Networks (DSN)*. 54–66. doi:10.1109/DSN53405.2022.00018
- [104] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 1–17.
- [105] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.
- [106] Maarten van Steen and Andrew S. Tanenbaum. 2017. *Distributed Systems* (3th ed.). Maarten van Steen. <https://www.distributed-systems.net/>
- [107] 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.
- [108] Shaokang Xie, Dakai Kang, Hanzheng Lyu, Jianyu Niu, and Mohammad Sadoghi. 2025. Fides: Scalable Censorship-Resistant DAG Consensus via Trusted Components. *arXiv preprint arXiv:2501.01062* (2025).
- [109] 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. doi:10.1145/3293611.3331591
- [110] 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. doi:10.1007/s11390-018-1839-y
- [111] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. 2019. GEM²-Tree: A Gas-Efficient Structure for Authenticated Range Queries in Blockchain. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 842–853. doi:10.1109/ICDE.2019.00080
- [112] 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.

Received October 2024; revised January 2025; accepted February 2025