

UC Davis

UC Davis Electronic Theses and Dissertations

Title

Towards High-Performance and Fair Byzantine Consensus: Concurrency, Speculation, Tail-Forking Resilience, and Transaction Ordering

Permalink

<https://escholarship.org/uc/item/6m68s5xz>

ISBN

9798189233548

Author

Kang, Dakai

Publication Date

2026-06-25

Peer reviewed|Thesis/dissertation

Towards High-Performance and Fair Byzantine Consensus: Concurrency, Speculation,
Tail-Forking Resilience, and Transaction Ordering

By

DAKAI KANG
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Mohammad Sadoghi, Chair

Aditya Thakur

Amanda Raybuck

Committee in Charge

2026

*To my parents,
for the freedom to find my own way.*

Contents

Abstract	vi
Acknowledgments	viii
Chapter 1. Introduction	1
1.1. Challenging Path from Stable Leader to Rotational Leaders	3
1.2. Rapid View Synchronization	4
1.3. Boosting Throughput with Concurrent Consensus	5
1.4. Speculation: Finalizing Blocks Earlier	5
1.5. Slotting: Giving Correct Leaders More Chances to Speak	6
1.6. Carry: Tail-Forking Resilience with Linear Complexity	6
1.7. A New Concern: Fair Transaction Ordering	7
1.8. Efficient Fair Ordering	8
1.9. Dissertation Organization	8
Chapter 2. SpotLess: Concurrent Rotational Consensus Made Practical through Rapid View Synchronization	10
2.1. Preliminaries	12
2.2. SpotLess Design Principles	13
2.3. Concurrent Consensus	28
2.4. Clients and Transaction Execution	31
2.5. Evaluation	32
2.6. Related Work	44
2.7. Conclusion	46
Chapter 3. HotStuff-1: Linear Consensus with One-Phase Speculation	47
3.1. Background and System Model	49
3.2. Speculation in Streamlined Protocols	50

3.3. Speculative Core	55
3.4. Streamlined Speculation	63
3.5. Slotting	65
3.6. Evaluation	74
3.7. Correctness Proofs	83
3.8. Speculation Safety and Rollback Examples	87
3.9. Related Work	93
3.10. Conclusion	95
Chapter 4. Carry the Tail in Consensus Protocols	96
4.1. Background	98
4.2. Carry	102
4.3. Correctness Proofs (continued)	107
4.4. Related Work	110
4.5. Conclusion	112
Chapter 5. FairDAG: Consensus Fairness over Multi-Proposer Causal Design	113
5.1. Background	115
5.2. System Model	118
5.3. Overview	120
5.4. FairDAG-AB	123
5.5. FairDAG-RL	128
5.6. Comparing Fairness Protocols	135
5.7. Correctness Proof	137
5.8. Evaluation	144
5.9. Related Work	149
5.10. Conclusion	152
Chapter 6. FEAT: Fair and Efficient Adversarial Transaction Ordering	153
6.1. Background	154
6.2. System Model	155
6.3. Problem Statement	156

6.4. FEAT: AVL-Tree-Based Fair-Ordering Algorithm	161
6.5. Correctness and Complexity Proof	167
6.6. Conclusion	174
Chapter 7. Conclusion	175
7.1. Summary of Contributions	175
7.2. Future Directions	176
Bibliography	178

Abstract

Consensus protocols are foundational to the reliability and security of large-scale distributed data management systems. The rapid success of blockchain technologies, notably Bitcoin and Ethereum, has driven significant research interest in consensus protocols. Unlike traditional distributed databases, which assume replicas are either correct or crashed, blockchain systems operate in a Byzantine environment where replicas can exhibit arbitrary, potentially malicious behaviors aimed at maximizing profit at the expense of others. This added complexity demands more sophisticated consensus mechanisms.

The first Byzantine fault tolerance (BFT) protocol, Practical Byzantine Fault Tolerance (PBFT), introduced over two decades ago, was a pioneering advance in distributed consensus. Despite its success, PBFT faces several challenges, including a complex and costly recovery mechanism, which has inspired substantial research into alternative protocols and optimizations. Recently, rotational consensus protocols, exemplified by HotStuff, have gained prominence by addressing some of PBFT’s limitations—particularly its quadratic-complexity recovery mechanism. However, they also introduce trade-offs, including lower throughput, higher latency, vulnerability to tail-forking attacks, and ambiguities in view synchronization. Meanwhile, as Decentralized Finance (DeFi) continues to grow, fair transaction ordering has emerged as a critical concern that existing protocols do not adequately address.

This dissertation introduces five contributions that advance the state of Byzantine consensus protocols along two dimensions: performance and fairness. First, we present SpotLess, a high-performance rotational concurrent consensus protocol featuring a rapid view synchronization mechanism that synchronizes every view with only $O(\Delta)$ worst-case latency, eliminating the need for costly threshold signatures in most cases and enabling efficient concurrent processing. Second, we introduce HotStuff-1, a rotational consensus protocol that incorporates speculation to allow replicas to send speculative responses one phase earlier, achieving the first speculative BFT protocol with linear communication complexity; HotStuff-1 further employs a slotting mechanism that allows correct leaders to propose multiple blocks per view, mitigating the impact of slow or Byzantine leaders. Third, we propose Carry, a practical drop-in mechanism for HotStuff-like protocols that resolves tail-forking attacks while preserving linear communication complexity, by treating replica votes as first-class citizens that transport knowledge of the last safe block forward. Fourth,

we present FairDAG, a fair transaction ordering framework built on multi-proposer DAG-based consensus, which removes the single-leader bottleneck present in prior fair-ordering protocols, tolerates a higher number of faulty replicas, and guarantees stronger fairness properties. Fifth, we introduce FEAT, a fair-ordering algorithm that establishes a tight lower bound on the comparison complexity of enforcing Adjacent-Comparator Consistency—a sufficient condition for batch-order-fairness—and matches this bound per round via an AVL-tree-based construction. Together, these contributions collectively advance the field by addressing key performance and fairness challenges in modern consensus protocols.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Mohamad Sadoghi, whose invaluable guidance, patience, and unwavering support carried me through my doctoral studies. Time and again I have been struck by his insight, and his rigor has profoundly shaped the way I approach research. My eyes are grateful for all the beautiful hiking trails he has led me along, though my legs may beg to differ.

I am also deeply grateful to the members of my dissertation committee, Prof. Aditya Thakur and Prof. Amanda Raybuck, whose thoughtful comments and constructive suggestions substantially improved this work.

Beyond my committee, I am immensely indebted to the professors and mentors who guided and collaborated on the research projects I pursued during my doctoral studies, Prof. Jeremie Decouchant, Prof. Tien Tuan Anh Dinh, Prof. Suyash Gupta, Prof. Jelle Hellings, Dr. Lefteris Kokoris-Kogias, Prof. Dahlia Malkhi, Prof. Jianyu Niu, and Dr. Alberto Sonnino, for generously sharing their expertise and broadening my perspective. My sincere thanks likewise go to my labmates and collaborators, Dr. Junchao Chen, Dr. Hanzheng Lyu, Dr. Benjamin Marsh, Dr. Ray Neiheiser, Dr. Sajjad Rahn timer, Dr. Alejandro Ranchal-Pedrosa, and Shaokang Xie, for the stimulating discussions, technical help, and companionship over these years.

To my friends, Boris, Ziteng, Junwa, and Fan, I owe a special thanks for a friendship that has carried on from our high-school and college years to this day, and that has been a steady source of support through my hardest moments.

My deepest gratitude goes to my parents. The greatest support and sacrifice a parent can make is not to pave the road ahead, but to grant the freedom to seek one's own direction in life, and that is the gift they have given me. This thesis is dedicated to them.

Finally, a doctoral journey is one of relentless judgment. Yet facing the scrutiny of peers and reviews takes far less courage than confronting the judgment of oneself. Only after recognizing, accepting, and owning the darkness, the inadequacy, the pretense, and the confusion within can one grow into a sincere and decent person. For walking beside me through that reckoning, and for gently helping me take off the invisible mask I had worn for too long, I owe my deepest thanks to someone who, for a time, shared my days, and who shaped me more profoundly than she may ever know.

CHAPTER 1

Introduction

The emergence of Bitcoin [126] and blockchain technology has renewed interest in consensus-based resilient database systems [39, 40, 55, 57, 107, 129, 134] that provide resilience to failures and manage data among fully-independent parties.

Most renowned blockchain systems, such as Bitcoin, Ethereum [33], and Solana [163], are permissionless blockchains. Permissionless blockchains place no restriction on the number of participants, who can join and leave freely. However, they are often criticized for high energy consumption, low throughput, and high latency. In contrast, permissioned blockchains, despite imposing certain restrictions on participants, achieve high throughput, low latency, and lower energy consumption. Thanks to these advantages, permissioned blockchains play an increasingly important role today [76].

Both permissioned and permissionless blockchains are distributed database systems that aim to reach consensus. To reach consensus, two essential properties must be guaranteed: **safety** and **liveness**. Safety requires that every data-storing replica within the system maintains a consistent ledger, while liveness ensures that new transactions can continually be added to the ledger. Ensuring safety and liveness is challenging: replicas must exchange messages over the network to reach consensus, but neither replicas nor the network always behave as expected. Moreover, when failures occur, it is difficult to distinguish replica failures from network failures.

In traditional distributed databases [106, 131], crashing is the only type of replica failure typically considered. However, due to its decentralized nature, blockchains face greater challenges, as replicas can exhibit Byzantine behavior. Byzantine replicas, named after the well-known Byzantine Generals Problem [107], can behave in arbitrary malicious ways, such as tampering with, repeating, withholding, or forging messages, crashing, equivocating, and even colluding [39, 134].

The presence of Byzantine replicas makes the consensus problem considerably harder. To address this challenge, at the core of permissioned blockchains lies the *Byzantine Fault Tolerant (BFT)* consensus protocol. The classic BFT protocol, Practical Byzantine Fault Tolerance (PBFT) [40],

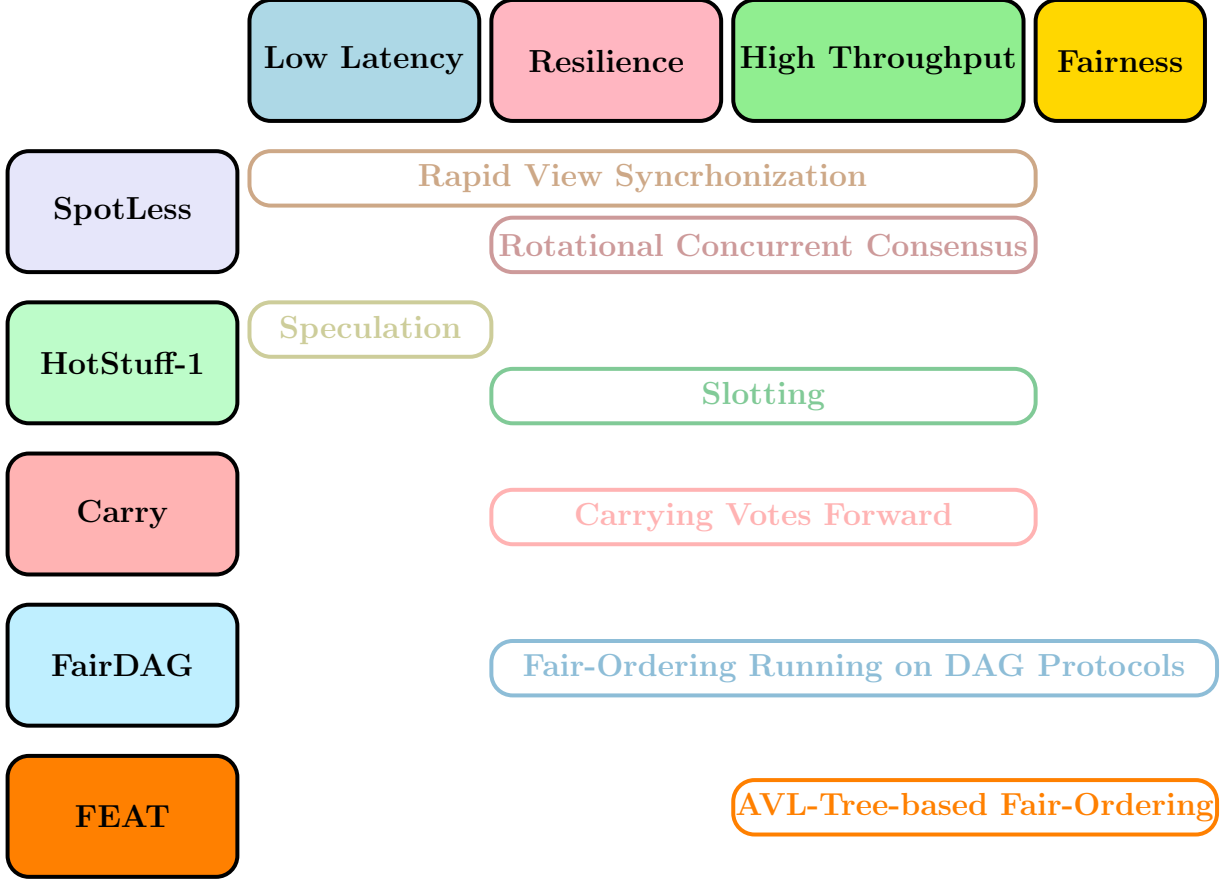


FIGURE 1.1. Optimization techniques used to implement high-performance and fair Byzantine fault tolerance protocols.

proposed over two decades ago, was the first practical BFT protocol that tolerates fewer than f Byzantine replicas out of n total replicas, where $n > 3f$, and reaches consensus under a partially synchronous network [55].

PBFT adopts a leader-based design with three communication phases. Although it is the first successful practical BFT consensus protocol, its various limitations have spurred ongoing improvements. To date, many innovative protocols [5, 42, 49, 66, 74, 77, 80, 82, 94, 102, 108, 118, 123, 136, 151, 164] have been proposed. While ensuring safety and liveness, these protocols employ novel strategies to improve system throughput and resilience to failures, and to reduce latency and complexity. The road of exploration, however, is not without obstacles: improvements in one aspect often come at the expense of others. Our goal is to approach an optimal solution that excels along all of these dimensions; although we may never fully reach it, we are continually moving closer.

In the following sections, beginning with the transition from stable-leader to rotational consensus protocols, we provide an overview of how our work contributes to the ongoing exploration of an optimal consensus protocol, which should possess the strengths mentioned above: resilience to failures, high throughput, low latency, and fair transaction ordering, the last being a new property that has recently drawn significant attention. As Figure 1.1 shows, our approaches leverage strategies and techniques such as Rapid View Synchronization, Rotational Concurrent Consensus, Speculation, Slotting, Carrying-Votes, Multi-proposer DAG Consensus, and an AVL-Tree-based Fair Ordering algorithm.

1.1. Challenging Path from Stable Leader to Rotational Leaders

The normal-case workflow of PBFT proceeds as follows: (1) the leader broadcasts a block to all other replicas; (2) replicas perform an all-to-all communication round, confirming that the leader does not equivocate and preparing to commit the block; (3) replicas conduct another all-to-all communication round, and once a majority have prepared the block, they commit it.

PBFT has been criticized for its quadratic communication complexity, which arises from this all-to-all communication. Some protocols [66, 74] introduce threshold signatures as certificates to achieve linear communication complexity. However, they do not address PBFT’s complicated recovery mechanism, the *view change*. As mentioned above, PBFT is a leader-based consensus protocol in which a well-performing leader is crucial for reaching consensus. Each view corresponds to one leader, and when a failure occurs, replicas move to the next view and the leader is replaced, a process known as view change. When a failure is detected, replicas must communicate to confirm its occurrence and reach consensus on the state prior to the failure, incurring quadratic communication overhead.

Rotational consensus protocols [4, 8, 60, 62, 89, 118, 154], represented by HotStuff [164], overcome this limitation and have led the research trend in recent years. Rotational consensus protocols proceed in views, in which replicas serve as the leader in a round-robin manner, with each leader allowed to propose only one block per view. Upon entering a new view, each replica sends the highest certificate it knows to the leader of that view, without notifying other replicas. The block proposed in the new view must extend the block carrying that highest certificate.

The view-change mechanism of rotational consensus offers two main advantages: (i) **Linear communication complexity**: since replicas advance to higher views autonomously, no all-to-all communication is required to trigger a view change; (ii) **Simplicity**: each replica sends only the highest certificate to the leader.

These advantages, however, come with the following costs: (D1) **View Synchronization Problem**: linear view changes can cause gaps in views among replicas due to faulty leaders or unreliable network, yet view synchronization is essential for liveness. (D2) **Low Throughput**: out-of-order processing, which is enabled in stable-leader protocols, is disabled in rotational consensus because each new block must extend the highest certificate. (D3) **Extra Latency**: HotStuff, the first rotational consensus protocol, requires one additional communication phase to ensure that the highest certificate is known to the new leader whenever it is already known to a majority of correct replicas. (D4) **Unavoidable Faulty Leaders**: because leaders rotate in a round-robin manner, the protocol must pass through views with faulty leaders, which may delay consensus and result in higher latency and lower throughput.

1.2. Rapid View Synchronization

For safety, replicas do not vote for blocks from lower or higher views. To ensure liveness, therefore, at least $2f + 1$ correct replicas must be in the same view as the leader; otherwise, no block can obtain a certificate.

The first rotational consensus protocol, HotStuff, relies on a black-box PaceMaker for view synchronization, which has been criticized as impractical. A significant body of research [42, 108, 109, 110, 115, 127, 128, 161] has since focused on designing practical view synchronization protocols. All such protocols, however, assume a Global Stabilization Time (GST), after which message delays are bounded by Δ and timer drift between replicas is negligible. These protocols divide views into epochs, each containing $O(f)$ views, and perform view synchronization at the beginning of every epoch. Under this design, view synchronization requires $O(f\Delta)$ time in the worst case. Additionally, to maintain linear communication complexity, these protocols rely on costly threshold signatures.

Our solution, SpotLess [94], introduces a practical *Rapid View Synchronization* mechanism with several advantages: (i) instead of performing view synchronization every epoch, SpotLess

synchronizes every view, incurring only $O(\Delta)$ latency in the worst case and remaining independent of GST; (ii) SpotLess avoids costly threshold signatures in most cases, using them only when absolutely necessary; and (iii) SpotLess integrates view synchronization directly into the consensus protocol, eliminating the need for a separate PaceMaker.

1.3. Boosting Throughput with Concurrent Consensus

In rotational consensus protocols, replicas cannot fully utilize their computing power and network resources because out-of-order processing is disabled. As a result, rotational consensus protocols generally achieve lower throughput than stable-leader consensus protocols.

Concurrent consensus [77, 151] offers a solution to this limitation. In leader-based protocols, the leader often becomes a performance bottleneck because it carries the heaviest workload. Concurrent consensus protocols were initially developed to address this single-leader bottleneck. In these protocols, multiple consensus instances (e.g., PBFT instances) operate in parallel, with each replica acting as the leader of one instance. Each instance maintains its own independent chain; committed blocks across chains are then globally ordered before execution, following a predetermined ordering rule.

Prior approaches have run concurrent instances of stable-leader consensus protocols. When a leader fails, however, complex recovery mechanisms are triggered, which can delay the global ordering and execution of committed blocks. In contrast, the concurrent consensus mechanism we propose in SpotLess runs multiple SpotLess instances in parallel, leveraging the simplicity of rotational consensus and providing greater resilience to faulty behaviors.

1.4. Speculation: Finalizing Blocks Earlier

Beyond view synchronization, another essential task during view change is state synchronization. In HotStuff, the first rotational consensus protocol, the leader alone is responsible for collecting states, which requires an additional phase to verify that it holds the highest state. This results in higher latency than protocols such as PBFT.

In recent years, various protocols [4, 8, 60, 62, 89, 154], including HotStuff-2 [118], have introduced solutions that remove this extra phase. To further reduce latency, we propose *HotStuff-1*, which incorporates speculation into HotStuff-2, allowing replicas to send speculative responses to clients one phase earlier.

Speculation itself is not new and has been used in several prior protocols [66,74,102]. However, previous speculative regimes incur quadratic complexity during view changes when failures occur. Our protocol, HotStuff-1 [93], is the first to implement speculation with linear communication complexity.

Applying speculation to HotStuff-2 is not without challenges. In HotStuff-1, we identify a *Prefix Speculation Dilemma* in which a client might mistakenly perceive a transaction as committed. To resolve this, we propose two necessary conditions—the *Prefix Speculation Rule* and the *No-Gap Rule*—that replicas must verify before speculatively executing a transaction.

1.5. Slotting: Giving Correct Leaders More Chances to Speak

Because leaders rotate in a round-robin manner, rotational consensus protocols must pass through views led by faulty leaders. To extract Maximal Extractable Value (MEV), a faulty leader may delay proposing its block until the timer nearly expires, causing the protocol to lose optimistic responsiveness (i.e., advancing at network speed) and resulting in reduced throughput and increased latency. Furthermore, in streamlined versions where phases of consecutive views overlap for higher throughput, a faulty leader can execute a tail-forking attack [65] by intentionally ignoring the certificate formed for the previous view’s block. Such tail-forking attacks not only reduce performance but can also prevent a correct replica’s block from ever being committed.

To address these issues, we introduce *slotting* into HotStuff-1. With slotting, each leader can propose multiple blocks (slots) within a single view before the timer expires. This mechanism gives correct leaders more opportunities to propose blocks, thereby enhancing system throughput and mitigating the impact of slow leaders and tail-forking attacks.

The introduction of slotting, however, raises a new challenge: the certificate for the last slot in a view may not reach all replicas before their timers expire. This complicates state synchronization for the next leader, even when the current leader is correct. To overcome this, we design *helper blocks* that can be included in the first slot of each view, ensuring smoother state synchronization across views.

1.6. Carry: Tail-Forking Resilience with Linear Complexity

While slotting mitigates the impact of faulty leaders, it does not fully prevent tail-forking attacks. In streamlined HotStuff-like (HS-like) protocols, replicas guard their locks but do not

guard their votes across view changes. A Byzantine leader can exploit this gap by ignoring the votes collected for the previous view’s block and proposing a new block that does not extend it, effectively forking the tail of the chain. Such tail-forking attacks can prevent a correct replica’s block from ever being committed, causing throughput to collapse [65].

Existing solutions to tail-forking, such as BeeGees [65], sacrifice the linear communication complexity that makes HS-like protocols attractive: they require an incoming leader either to collect a quorum of votes for the previous tail or to prove that no such quorum exists, which incurs quadratic overhead or relies on computationally prohibitive SNARKs.

Our solution, *Carry*, resolves this dilemma by treating replica votes as first-class citizens that not only help reach agreement but also transport the knowledge of the last safe block forward. A Byzantine leader cannot skip a safe block without producing a proof that it received no information about that block, and a correct future leader can reinstate a previous safe block even without collecting a full quorum of votes for it. Unlike prior approaches, *Carry* avoids quadratic blow-up: the leader only needs to justify the views it skips using the vote information it has collected, keeping all attachments bounded by $O(n \cdot \rho)$, where ρ is a small protocol parameter. We present *Carry-Tail*, a full atomic broadcast protocol that incorporates *Carry* into HotStuff-2, achieving tail-forking resilience while preserving the linear communication complexity characteristic of HS-like protocols.

1.7. A New Concern: Fair Transaction Ordering

Traditional BFT protocols prioritize safety and liveness. As the market capitalization of Decentralized Finance (DeFi) continues to grow, however, a new concern—*fairness*—has emerged as a critical property.

In leader-based consensus protocols, the leader has full control over the transaction order within each block. This enables a faulty leader to engage in manipulative practices such as front-running and back-running, altering the transaction order to generate profit at the expense of others.

Fair-ordering protocols [99, 100, 104, 169] have been developed to restrict the leader’s control over transaction ordering by requiring the leader to include local orderings from a majority of replicas within each block. The final order is then derived from the local orderings in committed blocks, reflecting the preferences of not only the leader but also other replicas. Under specific

conditions, this approach can guarantee fairness properties such as *Ordering Linearizability* and γ -*Batch-Order-Fairness*.

Existing fair-ordering protocols, however, are built on top of leader-based systems, where the leader can still selectively include the local orderings that most serve its interests. To further reduce the leader’s influence on the final transaction order, we introduce *FairDAG*, a fair-ordering protocol that operates atop a multi-proposer Directed Acyclic Graph (DAG) framework [49, 97, 149, 150]. By leveraging the properties of DAG-based consensus, FairDAG removes the single-leader bottleneck, is more resilient to faults, tolerates a higher number of faulty replicas, and guarantees $\frac{n}{2}$ -*correct-Batch-Order-Fairness*, a stronger fairness property that we introduce.

1.8. Efficient Fair Ordering

While FairDAG addresses the architectural limitations of leader-based fair-ordering protocols, a fundamental algorithmic question remains. All existing protocols that guarantee γ -*Batch-Order-Fairness* construct a complete pairwise precedence relation among transactions, incurring quadratic complexity in the size of the transaction set. Is this quadratic cost intrinsic to enforcing batch-order-fairness, or merely an artifact of current protocol designs?

We answer this question with *FEAT*, a fair-ordering algorithm that establishes the complexity-theoretic foundation for fair transaction ordering. We formalize fairness constraints as a global pairwise comparator over transactions and introduce a sufficient condition for γ -*Batch-Order-Fairness*, called *Adjacent-Comparator Consistency*. We prove that enforcing this condition requires $\Omega(|\mathcal{T}| \log |\mathcal{T}|)$ comparisons in the worst case, establishing a tight lower bound. FEAT achieves this optimal bound in every round via an AVL-tree-based construction [9], improving upon the quadratic complexity of all prior fair-ordering protocols and demonstrating that the overhead of fairness can be significantly reduced.

1.9. Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents *SpotLess*, a rotational concurrent consensus protocol with a rapid view synchronization mechanism that achieves high throughput while preserving low-cost recovery. Chapter 3 introduces *HotStuff-1*, which incorporates speculation and slotting into HotStuff-like protocols to reduce client-perceived latency and mitigate the impact of slow or Byzantine leaders. Chapter 4 proposes *Carry*, a drop-in mechanism

that resolves tail-forking attacks in HotStuff-like protocols while preserving linear communication complexity. Chapter 5 presents *FairDAG*, a fair-ordering framework built on Multi-proposer DAG-based consensus that mitigates the single-leader bottleneck and strengthens resilience against adversarial ordering manipulation. Chapter 6 introduces *FEAT*, which establishes a tight lower bound on the comparison complexity of enforcing Adjacent-Comparator Consistency, a sufficient condition for γ -Batch-Order-Fairness, and matches this bound per round via an AVL-tree-based construction. Finally, Chapter 7 concludes the dissertation with a summary of contributions and directions for future research.

CHAPTER 2

SpotLess: Concurrent Rotational Consensus Made Practical through Rapid View Synchronization

Our journey toward high-performance Byzantine consensus begins with SPOTLESS, a protocol that simultaneously attacks the two principal weaknesses of rotational consensus: the view-synchronization problem and the low throughput caused by disabled out-of-order processing. The material in this chapter appeared as [94].

Protocol	Environment		Concurrent Consensus	Streamlined Consensus	Threshold Signatures	Phases	Communication Complexity		
	Safety	Liveness					Messages	(at leader)	(per decision)
SPOTLESS	Asynchronous	Partial Synchrony	yes	yes	no	6	$c(3\mathbf{n}^2)$	$c(3\mathbf{n})$	$\mathbf{n}^2$
PBFT [40]	Asynchronous	Partial Synchrony	no	no	no	3	$2\mathbf{n}^2$	$3\mathbf{n}$	$2\mathbf{n}^2$
RCC [77]	Asynchronous	Partial Synchrony	yes	no	no	3	$c(2\mathbf{n}^2)$	$c(3\mathbf{n})$	$2\mathbf{n}^2$
HOTSTUFF [164]	Asynchronous	Partial Synchrony	no	yes	yes	8	$8\mathbf{n}$	$4\mathbf{n}$	$2\mathbf{n}$

FIGURE 2.1. Comparison of SPOTLESS with three state-of-the-art consensus protocols. Here, $\mathbf{n}$ is the number of replicas, c , $1 \leq c \leq \mathbf{n}$, is the number of concurrent instances, and the *per decision* cost is the amortized cost of a single consensus decision.

Two recent lines of work have addressed the limitations of leader-based consensus separately. *Concurrent consensus protocols* [77, 151, 152] run multiple PBFT-style instances in parallel—each replica acts as the leader of one instance—to remove the single-leader bottleneck. While this improves scalability and throughput, these protocols inherit PBFT’s heavyweight recovery: when a leader fails, complex view-change protocols are triggered, and recovery costs grow with the number of concurrent instances [77]. *Streamlined rotational consensus protocols*, exemplified by HOTSTUFF [164], replace the heavyweight view-change with linear-complexity leader rotation, significantly simplifying recovery. The simplicity comes at a price: by disabling out-of-order processing in protocols with stable leaders, rotational protocols are bottlenecked by message delays and achieve much lower throughput than PBFT. Furthermore, HOTSTUFF relies on a black-box Pacemaker for view synchronization [42, 108] and on costly threshold signatures to keep communication linear.

In this chapter, we present SPOTLESS, the first practical consensus protocol that combines simplicity with high performance. SPOTLESS does so by combining a novel *streamlined rotational consensus design* that is optimized toward simplicity, resilience, low message complexity, and latency with a high-performance *concurrent consensus architecture*. Central to the streamlined rotational consensus design of SPOTLESS is *Rapid View Synchronization* (RVS), which provides continuous low-cost leader rotation to deal with malicious behavior. RVS enables SPOTLESS to work in more general network assumptions, without a need for a *Global Synchronization Time* to synchronize views, and recover valid earlier views with the aid of non-faulty replicas without the need to rely on the leader.

The rotational design of SPOTLESS eliminates the need for the traditional complex and error-prone view-change protocols in PBFT and its variants: due to the rotational design of SPOTLESS, only information on a single round is used during recovery. In addition, RVS provides strong *view synchronization*, resolving the liveness issues of previous works. Furthermore, RVS does not require costly threshold signatures and provides robust failure recovery steps even when communication is unreliable. Finally, by combining the *streamlined rotational consensus design* with a *concurrent consensus architecture*, we remove the bottleneck of message delays typically seen in rotational designs *without* having to resort to highly-complex implementation techniques such as out-of-order processing.

To evaluate the performance of SPOTLESS in practice, we have implemented SPOTLESS in APACHE RESILIENTDB (Incubating), our high-performance resilient blockchain database that serves as a testbed for future RDMS technology. Our evaluation shows that SPOTLESS greatly outperforms existing consensus protocols such as PBFT [39, 40] by up to 430%, NARWHAL-HS [49] by up to 137%, and HOTSTUFF [164] by up to 3803%. Furthermore, due to the low message complexity of SPOTLESS, it is even able to outperform RCC [77] by up to 23% in normal conditions while serving client requests with lower latency in all cases. Finally, due to the robustness of RVS, SPOTLESS is able to maintain a stable latency and consistently high throughput even during failures.

Our contributions are as follows:

- (1) In Section 2.2, we present the single-instance *streamlined rotational consensus design* of SPOTLESS that provides the consensus replication using *rapid view synchronization*.

- (2) In Section 2.3, we provide the *concurrent consensus architecture* employed by SPOTLESS to run multiple instances of the streamlined rotational consensus instances *concurrently*, due to which SPOTLESS has highly-scalable throughput akin to RCC.
- (3) In Section 2.5, we empirically evaluate SPOTLESS in APACHE RESILIENTDB and compare its performance with state-of-the-art consensus protocols such as HOTSTUFF [164], PBFT [39, 40], RCC [77], and NARWHAL-HS [49]. In our evaluation, we show the *excellent properties* of SPOTLESS, which is even able to achieve higher throughput than the concurrent consensus protocol RCC, while providing a low and stable latency in all cases.

In addition, we introduce the terminology and notation used throughout this chapter in Section 2.1, discuss related work in Section 2.6, and conclude on our findings in Section 2.7. The properties of SPOTLESS and how they compare with other common and state-of-the-art consensus protocols are summarized in Figure 2.1.

2.1. Preliminaries

System We model our *system* as a fixed set of replicas $\mathfrak{R}$. We write $\mathbf{n} = |\mathfrak{R}|$ to denote the number of replicas and we write $\mathbf{f}$ to denote the number of *faulty replicas*. Each replica $R \in \mathfrak{R}$ has a unique identifier $\text{id}(R)$ with $0 \leq \text{id}(R) < \mathbf{n}$. We assume $\mathbf{n} > 3\mathbf{f}$ (a minimal requirement to provide consensus in an asynchronous environment [76]), that non-faulty replicas behave in accordance with the protocols they are executing, and that faulty replicas can behave arbitrarily, possibly coordinated and malicious ways. We do not make any assumptions about clients: all clients can be malicious without affecting SPOTLESS.

Consensus SPOTLESS is a *consensus protocol* that decides the sequence of *client requests* executed by all non-faulty replicas in the system $\mathfrak{R}$. To do so, SPOTLESS provides three *consensus guarantees* [76, 157]:

- (1) *Termination*. If non-faulty replica $R \in \mathfrak{R}$ decides upon an ρ -th client request, then all non-faulty replicas $Q \in \mathfrak{R}$ will decide upon an ρ -th client request;
- (2) *Non-Divergence*. If non-faulty replicas $R_1, R_2 \in \mathfrak{R}$ make ρ -th decisions τ_1 and τ_2 , respectively, then $\tau_1 = \tau_2$ (they decide upon the same ρ -th client request).
- (3) *Service*. Whenever a non-faulty client c requests execution of τ , then all non-faulty replicas will eventually decide on a client request of c .

We note that we use SPOTLESS in a setting of a replicated service that executes client requests. Hence, instead of the abstract *non-triviality guarantee* typically associated with consensus [76], SPOTLESS guarantees *service*. Adapting SPOTLESS to settings where other versions of *non-triviality* are required is straightforward.

Communication We assume *asynchronous communication*: messages can get lost or arbitrarily delayed. As consensus cannot be solved in asynchronous environments [57], we adopt the partial synchrony model of PBFT [39,40]: we *always* guarantee non-divergence (referred to as *safety*), while only guaranteeing termination and service during periods of reliable communication with a bounded message delay (referred to as *liveness*). We assume that periods of unreliable communication are always followed by sufficiently-long periods of synchronous communication (during which SPOTLESS can complete a limited number of steps to restore liveness).

Authentication We assume *authenticated communication*: faulty replicas are able to impersonate each other, but replicas cannot impersonate non-faulty replicas. Authenticated communication is a minimal requirement to deal with malicious behavior [76]. To enforce authenticated communication, we use two mechanisms: *message authentication codes* (MACs) and *digital signatures* (DSs) [95]. As MACs do not guarantee *tamper-free message forwarding*, we only use MACs (which are cheaper than DSs) to authenticate those messages that are not forwarded. For all other messages, we use DSs. We write $\langle v \rangle_p$ to denote a value v signed by participant p (a client or a replica). Finally, we write d_v to denote the message digest of a value v constructed using the same *secure cryptographic hash function* as the one used when signing v [95].

2.2. SpotLess Design Principles

SPOTLESS combines a streamlined rotational consensus design with a high-performance concurrent architecture. To maximize resilience in practical network environments in which communication can become *unreliable* and messages can get lost, the streamlined rotational consensus instances of SPOTLESS use *Rapid View Synchronization* (RVS) to assure that each instance can always recover and resume consensus.

Our presentation of individual SPOTLESS instance is broken up into five parts. Each *streamlined rotational consensus instance* of SPOTLESS operates in views $v \leftarrow 0, 1, 2, 3, \dots$. First, in Section 2.2.1, we show the two steps in every view. Second, in Section 2.2.2, we present the normal-case

replication steps and the three-phase commit algorithm used by each streamlined rotational consensus instance. Third, in Section 2.2.3, we formalize the guarantees provided by the normal-case replication steps and prove the safety of SPOTLESS. Then, in Section 2.2.4, we present the design of RVS. RVS bootstraps the guarantees provided by the normal-case replication toward providing per-instance consensus. Next, in Section 2.2.5, we describe how SPOTLESS assures per-instance consensus in an asynchronous environment and formally prove the liveness of SPOTLESS.

2.2.1. Steps in Every View: Propose and Synch Primitives. View v is coordinated by the replica $L \in \mathfrak{R}$ with $\text{id}(L) = v \bmod \mathbf{n}$. We say that L is the *leader* of view v and all other replicas act as *backups*. In view v , leader L will be able to *propose* the next client request τ upon which the system aims to achieve consensus. To do so, the system proceeds in *two steps i.e. Propose and Synch*. First, the leader inspects the existing chain and decides from which proposal it extends a new proposal, then the leader picks a valid client request τ , wraps and broadcasts a new proposal and broadcasts. Second, the backup replicas decide whether to vote for the new proposal and broadcast their decisions, where the new proposal is *conditionally prepared* by a replica if it receives $\mathbf{n} - \mathbf{f}$ concurring votes. Now, we explain the two steps in detail below.

First, leader L inspects the results of the preceding views to determine the highest extendable proposal $\mathbb{P}'$ past view $v - 1$, such that L believes that at least $\mathbf{n} - \mathbf{f}$ replicas will vote for a new proposal extending $\mathbb{P}'$. We will explore how $\mathbb{P}'$ is chosen in Section 2.2.3. Then, leader L picks a client request τ from some client c that it has not yet proposed and proposes τ by broadcasting a PROPOSE message of the form $\mathbb{P} := \text{PROPOSE}(v, \tau, \text{cert}(\mathbb{P}'))$ to all backups, in which $\text{cert}(\mathbb{P}')$ is a *certificate* for the preceding proposal $\mathbb{P}'$ that L chooses. The certificate is either a list of $\mathbf{n} - \mathbf{f}$ digital signatures, and we will explain how certificates are used in Section 2.2.3. To assure that τ cannot be forged by the leader, we assume that all client requests are digitally signed by the client c . To assure that the authenticity of $\mathbb{P}$ can be established and that $\mathbb{P}$ can be forwarded, the leader L will digitally sign the message $\mathbb{P}$.

Second, the backups establish whether the leader L correctly proposed a *unique* proposal to them. Specifically, the backups will exchange SYNC messages between them via which they can determine whether $\mathbb{P}$ is the only proposal that can collect enough endorsements to generate a certificate in the current view (necessary to provide non-divergence) and to ensure that enough non-faulty replicas received the same well-formed proposal $\mathbb{P}$ to assure that $\mathbb{P}$ can be recovered in any

future view (independent of any malicious behavior). To do so, each backup $R \in \mathfrak{R}$ performs the following steps upon receiving message $\mathbb{P} := \text{PROPOSE}(v, \tau, \text{cert}(\mathbb{P}'))$ with digital signature $(\mathbb{P}')_L$:

- (1) R checks whether $(\mathbb{P}')_L$ is a *valid digital signature*;
- (2) R checks whether τ is a *valid client request*;
- (3) R checks whether view v is the *current* view; and
- (4) R checks whether $\text{cert}(\mathbb{P}')$ is valid if R has not *conditionally prepared* $\mathbb{P}'$.

Only if the proposal $\mathbb{P}$ passes all these checks, the replica R will consider $\mathbb{P}$ to be *well-formed*. In this case, backup R *records* $\mathbb{P}$. If $\mathbb{P}$ is the first *acceptable* proposal R receives in view v (we detail the conditions of *acceptable* proposals in Section 2.2.3), R will broadcast the message $ms_R := \text{SYNC}(v, \text{claim}(\mathbb{P}), \mathbb{CP})$, in which $\text{claim}(\mathbb{P}) := (v, d_{\mathbb{P}}, (\mathbb{P})_L)$ is a *claim* that $\mathbb{P}$ is the well-formed proposal that backup R received in view v , and $\mathbb{CP}$ is a set of pairs in the form of (view, digest) for the proposals that R has *conditionally prepared*. We will explore the details of $\mathbb{CP}$ in Section 2.2.3.

Otherwise, if backup R determines a failure in view v (R did not receive any valid proposals in view v while it should have received one), then R will end up broadcasting the message $ms_R := \text{SYNC}(v, \text{claim}(\emptyset), \mathbb{CP})$ to all backup replicas, claiming to have not received any valid proposals in view v .

To assure that the authenticity of ms_R can be established *without verifying digital signatures* and that ms_R can be forwarded, the replica R will include both a message authentication code and the digital signature $(ms_R)_R$. To reduce computational costs, the message authentication codes of SYNC messages are *always* verified, whereas digital signatures are *only verified in cases where recovery is necessary*, we refer to Section 2.2.4 for the exact verification rules.

REMARK 2.2.1. *To simplify presentation, all replicas broadcast messages to all replicas (including themselves). Doing so, we simplify the claims and proofs made in this chapter. Without affecting the correctness of SPOTLESS, one can eliminate sending messages to oneself. In addition, the leader broadcasts the proposal $\mathbb{P}$ together with a matching SYNC message.*

2.2.2. Streamlined Three-Phase Commit Algorithm in Normal Case. SPOTLESS adopts a three-phase commit algorithm. Each phase takes one view, and SPOTLESS rotates the leader view by view, to eliminate complex failure detection and recovery. A successful first phase of SPOTLESS

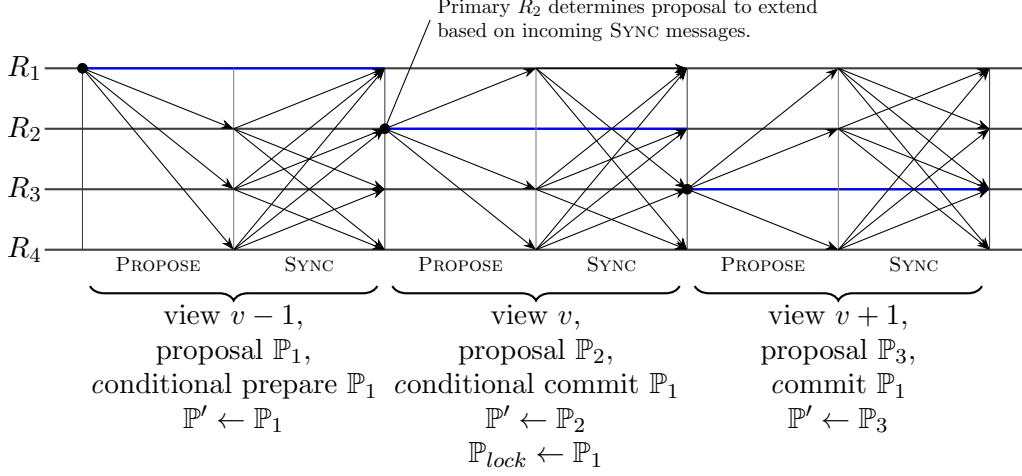


FIGURE 2.2. A schematic representation of the normal-case replication protocol in a streamlined rotational consensus instance of SPOTLESS in three consecutive views $v - 1$ (leader R_1), v (leader R_2), and $v + 1$ (leader R_3). $\mathbb{P}'$ refers to the *highest extendable proposal*. We have *not* visualized those messages one can eliminate (see Remark 2.2.1).

establishes a *conditional prepare* that ensures non-divergence of proposals within the view; a successful second phase of SPOTLESS establishes a *conditional commit*; and a successful third phase achieves a *commit* that ensures the preservation of proposals across views. We explain the conditions to establish these three proposal states (i.e., *conditional prepare*, *conditional commit*, and *commit*) later in Definition 2.2.2. In Figure 2.2, we have visualized three *consecutive* operations of a streamlined rotational consensus instance in views $v - 1$, v , and $v + 1$ with respect to a proposal $\mathbb{P}$ and in Figure 2.3, we present the pseudo-code of the normal-case operations of the streamlined rotational consensus.

The normal-case replication protocol of SPOTLESS only establishes a minimal guarantee on the overall state of the system that can be proven with a straightforward quorum-based argument [76]:

THEOREM 2.2.1. *Consider view v of the normal-case replication of a streamlined rotational consensus instance of SPOTLESS and consider two replicas $R_1, R_2 \in \mathfrak{R}$. If $\mathbf{n} > 3\mathbf{f}$ and replica R_i , $i \in \{1, 2\}$, receives a set of authenticated messages $\{\text{SYNC}(v, \text{claim}(\mathbb{P}_{i,Q})) \mid Q \in \mathfrak{Q}_i\}$ from a set $\mathfrak{Q}_i \subseteq \mathfrak{R}$ of $|\mathfrak{Q}_i| = \mathbf{n} - \mathbf{f}$ replicas, and all $\text{claim}(\mathbb{P}_{i,Q})$, $i \in \{1, 2\}$, represent proposal $\mathbb{P}_i$, then $\mathbb{P}_1 = \mathbb{P}_2$.*

PROOF. We prove the theorem by contradiction. Assume $\mathbb{P}_1 \neq \mathbb{P}_2$. Replica R_i received claims representing $\mathbb{P}_i$ from the $\mathbf{n} - \mathbf{f}$ in $\mathfrak{Q}_i$. As there are at most $\mathbf{f}$ faulty replicas, at-least $\mathbf{n} - 2\mathbf{f}$ replicas in $\mathfrak{Q}_i$ are non-faulty. Let $S_i \subseteq \mathfrak{Q}_i$ be all non-faulty replicas in $\mathfrak{Q}_i$.

As the non-faulty replicas in S_1 made claims representing $\mathbb{P}_1$, the non-faulty replicas in S_2 made claims representing $\mathbb{P}_2$, $\mathbb{P}_1 \neq \mathbb{P}_2$, and non-faulty replicas only make a single claim in view v (see Figure 2.3), we must have $S_1 \cap S_2 = \emptyset$. Hence, we have $|S_1 \cup S_2| \geq 2(\mathbf{n} - 2\mathbf{f})$. As all replicas in $S_1 \cup S_2$ are non-faulty, we also have $|S_1 \cup S_2| \leq \mathbf{n} - \mathbf{f}$. Hence, $2(\mathbf{n} - 2\mathbf{f}) \leq \mathbf{n} - \mathbf{f}$ must hold, which implies $\mathbf{n} \leq 3\mathbf{f}$, a contradiction. Hence, by contradiction, we conclude $\mathbb{P}_1 = \mathbb{P}_2$. $\square$

2.2.3. Rules Guaranteeing Safety. Beside the minimal guarantee in Theorem 2.2.1, SPOT-LESS ensures the safety of the system via a set of rules that non-faulty replicas must follow. In the meantime, the rules play an important role in helping restore liveness while ensuring safety. Before exploring the rules, we first introduce the necessary terminology.

DEFINITION 2.2.2. Let $\mathbb{P} := \text{PROPOSE}(v, \tau, \text{cert}(\mathbb{P}'))$ be a well-formed proposal in view v . We say that $\mathbb{P}'$ is the preceding proposal of $\mathbb{P}$. For any two proposals $\mathbb{P}_1$ and $\mathbb{P}_2$, we say that $\mathbb{P}_1$ precedes $\mathbb{P}_2$ if $\mathbb{P}_1$ is the preceding proposal of $\mathbb{P}_2$ or if there exists a proposal $\mathbb{P}^*$ such that $\mathbb{P}_1$ precedes $\mathbb{P}^*$ and $\mathbb{P}^*$ is the preceding proposal of $\mathbb{P}_2$. Let $\text{precedes}(\mathbb{P})$ be the set of all proposals that precede $\mathbb{P}$. The depth of proposal $\mathbb{P}$ is defined by $\text{depth}(\mathbb{P}) = |\text{precedes}(\mathbb{P})|$.

We say that a replica records $\mathbb{P}$ if it determines that $\mathbb{P}$ is well-formed (Line 15 in Figure 2.3) and say that it accepts $\mathbb{P}$ if it broadcasts SYNC messages with $\text{claim}(\mathbb{P})$.

We say that a replica conditionally prepares $\mathbb{P}$ if the replica received $\mathbb{P}$ and, during view v , the replica receives $\mathbf{n} - \mathbf{f}$ concurring votes for $\mathbb{P}$, i.e. a set of messages $\{\text{SYNC}(v, \text{claim}(\mathbb{P}_Q), \mathbb{C}\mathbb{P}) \mid Q \in \mathfrak{Q}\}$ with $\mathbb{P}_Q = \mathbb{P}$ from a set $\mathfrak{Q} \subseteq \mathfrak{R}$ of $|\mathfrak{Q}| = \mathbf{n} - \mathbf{f}$ replicas. We say that a replica conditionally commits $\mathbb{P}$ if, in a future view $w > v$, the replica conditionally prepares a proposal of the form $\mathbb{P}' := \text{PROPOSE}(w, \tau', \text{cert}(\mathbb{P}))$ that extends $\mathbb{P}$. We say that a replica locks $\mathbb{P}$ if $\mathbb{P}$ is the highest proposal that it conditionally commits, denoted by $\mathbb{P}_{\text{lock}}$. Also, we say that a replica commits $\mathbb{P}$ if, in a future view $u > v$, the replica conditionally prepares a proposal of the form $\mathbb{P}'' := \text{PROPOSE}(u, \tau'', \text{cert}(\mathbb{P}'))$ that extends $\mathbb{P}'$, with $u = w + 1 = v + 2$. We say that two proposals are conflicting if the preceding proposals of these two proposals are disjoint.

1: Let $L \in \mathfrak{R}$ be the replica with $v = \text{id}(L) \bmod \mathbf{n}$ (the leader).

2: **function** ACCEPTABLE($\mathbb{P} := \text{PROPOSE}(v, \tau, \text{cert}(\mathbb{P}'))$) **do**

3: Let $\mathbb{P}_{\text{lock}}$ be the highest proposal this replica conditionally committed.

4: **return** R conditionally prepared $\mathbb{P}'$ **and** $(v' < v_{\text{lock}}$ **or** $\mathbb{P}_{\text{lock}} \in \{\mathbb{P}'\} \cup \text{precedes}(\mathbb{P}'))$.

5: **function** HIGHESTEXTENDABLE() **do**

6: **for** v from *CurrentView* down to 0 **do**

7: **if** L conditionally prepared proposal $\mathbb{P}'$ of view v **then**

8: **if** L has a valid $\text{cert}(\mathbb{P}')$ **then**

9: **return** $\mathbb{P}', \text{cert}(\mathbb{P}')$.

10: **else if** L receives $\text{SYNC}(v_i, \text{claim}(\cdot)_i, \mathbb{CP}_i)$ from $\mathbf{n} - \mathbf{f}$ replicas with $\text{claim}(\mathbb{P}') \in \mathbb{CP}_i$ **then**

11: **return** $\mathbb{P}', \text{claim}(\mathbb{P}')$.

Leader role (running at L of view v) :

12: $\mathbb{P}, cc := \text{HIGHESTEXTENDABLE}()$; await client request $(\tau)_c$; broadcast $\text{PROPOSE}(v, \tau, cc)$.

Backup role (running at each $R \in \mathfrak{R}$) :

13: **event** R receives a well-formed proposal $\mathbb{P}$ **do**

14: **if** R has not sent SYNC in view v **and** $\text{ACCEPTABLE}(\mathbb{P})$ **then**

15: Broadcast $\text{SYNC}(v, \text{claim}(\mathbb{P}), \mathbb{CP})$.

16: **event** R determines a failure in view v **do**

17: Broadcast $\text{SYNC}(v, \text{claim}(\emptyset), \mathbb{CP})$.

18: **event** R receives $\text{SYNC}(v, \text{claim}(\mathbb{P}), \mathbb{CP})$ from $\mathbf{n} - \mathbf{f}$ replicas **do**

19: Conditionally prepare $\mathbb{P}$.

20: **event** R receives $\text{SYNC}(v', \text{claim}(\cdot), \mathbb{CP})$ from $\mathbf{f} + 1$ replicas with $v' > v$ and $\text{claim}(\mathbb{P}) \in \mathbb{CP}$ **do**

21: Conditionally prepare $\mathbb{P}$.

22: **event** R receives $\text{SYNC}(v, \text{claim}(\mathbb{P}), \mathbb{CP})$ from $\mathbf{f} + 1$ replicas **do**

23: **if** R has not sent SYNC in view v **then**

24: Broadcast $\text{SYNC}(v, \text{claim}(\mathbb{P}), \mathbb{CP})$.

25: **if** R does not know $\mathbb{P}$ **then**

26: Send $\text{ASK}(v, \text{claim}(\mathbb{P}))$ to the $\mathbf{f} + 1$ replicas.

27: **event** R receives $\text{ASK}(v, \text{claim}(\mathbb{P}))$ from R' **and** has recorded $\mathbb{P}$ **do**

28: Send $\mathbb{P}$ to R' .

FIGURE 2.3. The replication protocol in a SPOTLESS instance.

The proposal states *conditionally committed* and *committed* are analogous to the proposal states *prepared* and *committed* in traditional non-streamlined protocols such as PBFT [40]. In Figure 2.2, we show how a proposal establishes the three states in the normal case. Using this terminology, we can specify the safety guarantee that individual SPOTLESS instances will maintain

on the system: *no two conflicting proposals $\mathbb{P}_i$ and $\mathbb{P}_j$ can be both committed, each by a non-faulty replica*, which we will prove in Theorem 2.2.3.

To guarantee safety, central to the design principle are the rules that non-faulty replicas follow when deciding whether to *extend*, *accept*, or *conditionally prepare* a proposal.

The leader L can construct a *certificate* for proposal $\mathbb{P}$ after L *recorded* $\mathbb{P}$ and received $\mathbf{n} - \mathbf{f}$ SYNC messages with valid signatures for $\mathbb{P}$, i.e.

$$S = \{\text{SYNC}(v - 1, \text{claim}(\mathbb{P}_Q), \mathbb{CP}) \mid Q \in \mathfrak{Q}\}$$

with $\mathbb{P}_Q = \mathbb{P}$ and a valid signature from a set $\mathfrak{Q} \subseteq \mathfrak{R}$ of $|\mathfrak{Q}| = \mathbf{n} - \mathbf{f}$ replicas. The set S will be used to construct the certificate. Even if R fails to receive sufficient SYNC messages to *conditionally prepare* $\mathbb{P}$ in view $v - 1$, R will *conditionally prepare* $\mathbb{P}$ if it receives a valid *certificate* $\text{cert}(\mathbb{P})$.

Each SYNC message includes $\mathbb{CP}$ that consists of the views and digests of the sender's $\mathbb{P}_{\text{lock}}$ and all *conditionally prepared* proposals with a higher view than the view v_{lock} of proposal $\mathbb{P}_{\text{lock}}$:

$$\mathbb{CP} := \{v_{\mathbb{P}}, d_{\mathbb{P}} \mid \mathbb{P} \text{ is conditionally prepared} \wedge v_{\text{lock}} \leq v_{\mathbb{P}}\}.$$

R *conditionally prepares* $\mathbb{P}$ if it receives a set S' of SYNC messages from $\mathbf{f} + 1$ replicas claiming to have *conditionally prepared* $\mathbb{P}$, which implies at least one non-faulty replicas have *conditionally prepared* $\mathbb{P}$ after receiving $\mathbf{n} - \mathbf{f}$ concurring votes, where

$$S' = \{\text{SYNC}(w_{Q'}, \text{claim}(\mathbb{P}_{Q'}), \mathbb{CP}_{Q'}) \mid Q' \in \mathfrak{Q}'\}$$

with $w_{Q'} \geq v$, $\mathbb{P} \in \mathbb{CP}_{Q'}$, and $\mathfrak{Q}' \subseteq \mathfrak{R}$ with $|\mathfrak{Q}'| = \mathbf{f} + 1$.

A non-faulty leader of view v considers a proposal $\mathbb{P}'$ to be *extendable* if either of the following conditions is met:

E1 L has a valid *certificate* for $\mathbb{P}'$;

E2 L has received a set of SYNC messages from $\mathbf{n} - \mathbf{f}$ replicas that claim to have *conditionally prepared* $\mathbb{P}'$, i.e.

$$\{\text{SYNC}(w_Q, \text{claim}(\mathbb{P}_Q), \mathbb{CP}_Q) \mid Q \in \mathfrak{Q}\}$$

with $w_Q < v$ and $\mathbb{P}' \in \mathbb{CP}_Q$ and $\mathfrak{Q} \subseteq \mathfrak{R}$ with $|\mathfrak{Q}| = \mathbf{n} - \mathbf{f}$.

The leader L backtracks to earlier views to find the highest *extendable* proposal $\mathbb{P}'$ and then sets the preceding proposal to $\mathbb{P}'$. If $\mathbb{P}'$ satisfies E1, then L broadcasts $\mathbb{P} := \text{PROPOSE}(v, \tau, \text{cert}(\mathbb{P}'))$. Otherwise, L broadcasts $\mathbb{P} := \text{PROPOSE}(v, \tau, \text{claim}(\mathbb{P}'))$.

When receiving a well-formed new proposal of the form $\mathbb{P} := \text{PROPOSE}(v, \tau, \text{cert}(\mathbb{P}'))$ or $\text{PROPOSE}(v, \tau, \text{claim}(\mathbb{P}'))$, a replica R determines whether to *accept* $\mathbb{P}$ based on the following rules:

A1 Validity Rule: R has *conditionally prepared* $\mathbb{P}'$.

A2 Safety Rule: $\mathbb{P}'$ extends R 's locked proposal $\mathbb{P}_{\text{lock}}$, i.e.

$$\mathbb{P}_{\text{lock}} \in (\{\mathbb{P}'\} \cup \text{precedes}(\mathbb{P}')).$$

A3 Liveness Rule: $\mathbb{P}'$ has a higher view than $\mathbb{P}_{\text{lock}}$.

If A1 holds and either A2 or A3 holds, then R broadcasts $\text{SYNC}(v, \text{claim}(\mathbb{P}), \mathbb{CP})$. Otherwise, R keeps waiting for a proposal satisfying the acceptance requirement until its timer expires.

Due to unreliable communication or faulty behavior, non-faulty replica R may fail to receive any *acceptable* proposal from leader L_v but receive a set M' consisting of $\mathbf{f} + 1$ SYNC messages with the same $\text{claim}(\mathbb{P})$, formally, R receives

$$M' = \{\text{SYNC}(v, \text{claim}(\mathbb{P}_{Q^M}), \mathbb{CP}_{Q^M}) \mid Q^M \in \mathfrak{Q}^{\mathfrak{M}}\}$$

with $\mathbb{P}_{Q^M} = \mathbb{P}$ from a set $\mathfrak{Q}^{\mathfrak{M}} \subseteq \mathfrak{R}$ of $|\mathfrak{Q}^{\mathfrak{M}}| = \mathbf{f} + 1$ replicas. For easier restoration of liveness, SPOTLESS allows R to broadcast $\text{SYNC}(v, \text{claim}(\mathbb{P}), \mathbb{CP})$ if R considers $\mathbb{P}$ as *acceptable*.

In such a case, R is unaware of the full information of $\mathbb{P}$ and needs to catch up. To do so, R sends $\mathbf{a} := \text{ASK}(v, \text{claim}(\mathbb{P}))$ to the $\mathbf{f} + 1$ replicas in $\mathfrak{Q}^{\mathfrak{M}}$. After a good replica $R' \in \mathfrak{Q}^{\mathfrak{M}}$ receives $\mathbf{a}$, the replica R' will forward $\mathbb{P}$ to R if it has *recorded* a well-formed $\mathbb{P}$. To reduce the overhead of this mechanism in practical implementations, replicas can choose to first send ASK messages to replicas they already trust (e.g., based on previous behavior).

Based on the design principles above, we can prove the safety property of SPOTLESS step by step:

LEMMA 2.2.2.1. *If a non-faulty replica R conditionally prepares $\mathbb{P} = \text{PROPOSE}(v, \tau, \text{cert}(\mathbb{P}'))$, then for each proposal $\mathbb{P}^* \in \text{precedes}(\mathbb{P})$ that precedes $\mathbb{P}$, at least $\mathbf{n} - 2\mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas have conditionally prepared $\mathbb{P}^*$ and sent SYNC messages with $\mathbb{P}^* \in \mathbb{CP}$.*

PROOF. Assume that R is the first non-faulty replica that *conditionally prepares* $\mathbb{P}$. The only way for R to *conditionally prepare* $\mathbb{P}$ is to receive $\mathbf{n} - \mathbf{f}$ SYNC messages of the form $\text{SYNC}(v, \text{claim}(\mathbb{P}), \mathbb{CP})$, of which $\mathbf{n} - 2\mathbf{f} \geq \mathbf{f} + 1$ are from non-faulty replicas. Hence, at least $\mathbf{n} - 2\mathbf{f}$ non-faulty replicas must have *conditionally prepared* the preceding proposal $\mathbb{P}'$ of $\mathbb{P}$, as non-faulty replica only accept $\mathbb{P}$ if they *conditionally prepared* $\mathbb{P}'$. From the definition of $\mathbb{CP}$ we know that these $\mathbf{n} - 2\mathbf{f}$ non-faulty replicas must have send SYNC messages with $\mathbb{P}' \in \mathbb{CP}$ after they *conditionally prepare* $\mathbb{P}'$. Next, we can bootstrap this argument to show that the above statement holds for each proposal $\mathbb{P}^* \in \text{precedes}(\mathbb{P})$. $\square$

Using Lemma 2.2.2.1, we are able to prove safety:

THEOREM 2.2.3. *No two non-faulty replicas can commit conflicting proposals $\mathbb{P}_i$ and $\mathbb{P}_j$.*

PROOF. We shall prove this theorem by contradiction. Assume that both $\mathbb{P}_i$ and $\mathbb{P}_j$ are *committed* by (possibly distinct) non-faulty replicas and that $\mathbb{P}_i$ and $\mathbb{P}_j$ are conflicting. Without loss of generality, we can assume that proposal $\mathbb{P}_i$ is from view v_i , proposal $\mathbb{P}_j$ is from view v_j , and $\text{depth}(\mathbb{P}_j) \geq \text{depth}(\mathbb{P}_i)$. As $\mathbb{P}_i$ is *committed*, there must exist proposals $\mathbb{P}_{i+1}$ and $\mathbb{P}_{i+2}$ of the form

$$\mathbb{P}_{i+1} = \text{PROPOSE}(v_i + 1, \tau_{i+1}, \text{cert}(\mathbb{P}_i)); \text{ and}$$

$$\mathbb{P}_{i+2} = \text{PROPOSE}(v_i + 2, \tau_{i+2}, \text{cert}(\mathbb{P}_{i+1}))$$

that have been *conditionally prepared* by non-faulty replicas. Likewise, as $\mathbb{P}_j$ is *committed*, there must exists proposals $\mathbb{P}_{j+1}$ and $\mathbb{P}_{j+2}$ that have been *conditionally prepared* by non-faulty replicas.

Since $\mathbb{P}_j$ conflicts $\mathbb{P}_i$ and $\text{depth}(\mathbb{P}_j) \geq \text{depth}(\mathbb{P}_i)$, there must exists a depth d and proposals $\mathbb{P}_{d,i} \in \{\mathbb{P}_i\} \cup \text{precedes}(\mathbb{P}_i)$ and $\mathbb{P}_{d,j} \in \{\mathbb{P}_j\} \cup \text{precedes}(\mathbb{P}_j)$ such that $\text{precedes}(\mathbb{P}_{d,i}) = \text{precedes}(\mathbb{P}_{d,j})$ and $\mathbb{P}_{d,i} \neq \mathbb{P}_{d,j}$ (hence, the first proposals that precede $\mathbb{P}_i$ and $\mathbb{P}_j$, respectively, that are in conflict). As both $\mathbb{P}_i$ and $\mathbb{P}_j$ are *committed*, some non-faulty replicas must have *conditionally prepared* $\mathbb{P}_{d,i}$ and $\mathbb{P}_{d,j}$.

From Lemma 2.2.2.1 and the fact that $\mathbb{P}_{i+2}$ and $\mathbb{P}_{j+2}$ are *conditionally prepared* by a non-faulty replica, we conclude that $\mathbf{n} - 2\mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas *conditionally prepared* all proposals in $\text{precedes}(\mathbb{P}_{i+2})$ and in $\text{precedes}(\mathbb{P}_{j+2})$. Let $\mathbb{P}' \in \text{precedes}(\mathbb{P}_{j+2})$ be the proposal with $\text{depth}(\mathbb{P}_{i+1}) = \text{depth}(\mathbb{P}')$. By construction, $\mathbb{P}_d \in \text{precedes}(\mathbb{P}')$. hence, $\mathbb{P}'$ and $\mathbb{P}_{i+1}$ are in conflict and are both *conditionally prepared* by $\mathbf{n} - 2\mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas. Hence, by a similar

quorum argument as used to prove Theorem 2.2.1, we conclude there must be at least one non-faulty replica that *conditionally prepared* both $\mathbb{P}'$ and $\mathbb{P}_{i+1}$. This non-faulty replica must have also locked on either $\mathbb{P}_i$ (when it accepted $\mathbb{P}_{i+1}$) or the proposal $\mathbb{P}''$ preceding $\mathbb{P}'$ (when it accepted $\mathbb{P}'$). By construction, $\mathbb{P}'' \neq \mathbb{P}'$. Hence, one of these accept steps would violate the safety rule A2, a contradiction. $\square$

Theorem 2.2.3 proves SPOTLESS can ensure safety if we have *three-consecutive-view* requirement for committing a proposal. Next, we illustrate the necessity of the *three-consecutive-view* requirement (over a two-consecutive-view requirement) for SPOTLESS.

EXAMPLE 2.2.4. Assume a system with $\mathbf{n} = 3\mathbf{f} + 1$ replicas and that all replicas have conditionally prepared $\mathbb{P}_0$. We describe events that happen in the next six views to illustrate how two conflicting proposals can get committed if we relax the three-consecutive-view requirement.

- (1) Primary L_1 broadcasts proposal $\mathbb{P}_1$ extending $\mathbb{P}_0$. All replicas accept $\mathbb{P}_1$ and then conditionally prepare $\mathbb{P}_1$.
- (2) Primary L_2 broadcasts proposal $\mathbb{P}_2$ extending $\mathbb{P}_0$. All replicas accept $\mathbb{P}_2$ and then conditionally prepare $\mathbb{P}_2$.
- (3) Faulty leader L_3 sends proposal $\mathbb{P}_3$ extending $\mathbb{P}_2$ to $\mathbf{f} + 1$ non-faulty replicas including R_0 and sends proposal $\mathbb{P}'_3$ extending $\mathbb{P}_2$ to the other $\mathbf{f}$ non-faulty replicas. All $\mathbf{f}$ faulty replicas only send SYNC messages with $\text{claim}(\mathbb{P}_3)$ to R_0 , due to which only R_0 conditionally prepares $\mathbb{P}_3$, while other non-faulty replicas cannot conditionally prepare it.
- (4) L_4 broadcasts proposal $\mathbb{P}_4$ extending $\mathbb{P}_1$. All non-faulty replicas except R_0 conditionally prepare $\mathbb{P}_4$.
- (5) Faulty leader L_5 sends proposal $\mathbb{P}_5$ extending $\mathbb{P}_4$ to $\mathbf{f} + 1$ non-faulty replicas including R_1 and not including R_0 and sends proposal $\mathbb{P}_5$ extending $\mathbb{P}_4$ to the other $\mathbf{f}$ non-faulty replicas. All $\mathbf{f}$ faulty replicas only send Sync messages with $\text{claim}(\mathbb{P}_5)$ to R_1 , due to which only R_1 conditionally prepares $\mathbb{P}_5$ and commits $\mathbb{P}_1$, while others cannot conditionally prepares $\mathbb{P}_5$.
- (6) L_6 broadcasts proposal $\mathbb{P}_6$ extending $\mathbb{P}_3$ and all replicas except R_1 broadcast SYNC messages with $\text{claim}(\mathbb{P}_6)$. Then, all non-faulty replicas except R_1 conditionally prepare $\mathbb{P}_6$ and commit $\mathbb{P}_2$ that conflicts with $\mathbb{P}_1$, which was committed by R_1 .

In this setting, using a two-consecutive-view requirement for committing, $\mathbb{P}_1$ and $\mathbb{P}_2$ are conflicting proposals but are both committed by non-faulty replicas.

2.2.4. Bootstrapping Liveness with Rapid View Synchronization. *Rapid View Synchronization* (RVS) bootstraps the guarantees provided by normal-case replication toward providing consensus. RVS does so by dealing with asynchronous communication and by strengthening the guarantees on proposals of preceding views. In specific, the main services provided by RVS are a best-effort and quick *view synchronization* to assure that replicas end up in the same views whenever communication is sufficiently reliable and *low-cost state recovery* to enable cheap leader rotation to deal with failures of previous leaders.

To enable *Rapid View Synchronization*, for each view v , a replica must go through three states one by one:

ST1 Recording: waiting for a well-formed $\mathbb{P}$ that satisfies A1 and either A2 or A3 until state timer t_R expires;

ST2 Syncing: waiting for a set of SYNC messages with view v from a set $\mathfrak{Q} \subseteq \mathfrak{R}$ with $|\mathfrak{Q}| = \mathbf{n} - \mathbf{f}$ replicas;

ST3 Certifying: waiting for a set of messages

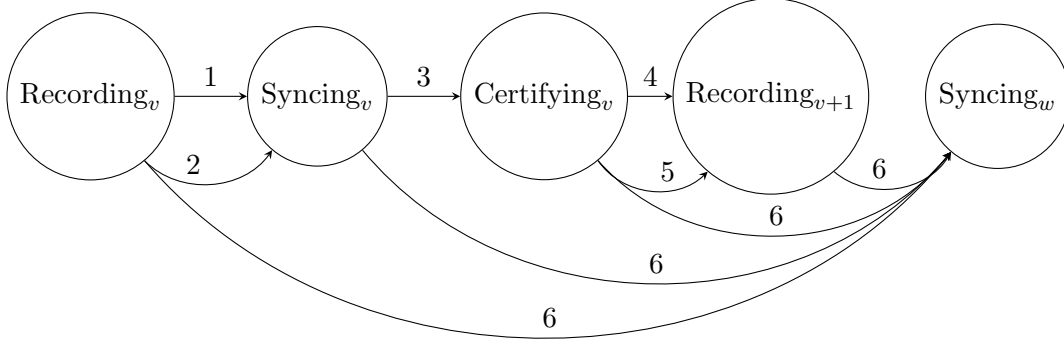
$$S = \{\text{SYNC}(v, \text{claim}(\mathbb{P}_{Q^A}), \mathbb{CP}) \mid Q^A \in \mathfrak{Q}^{\mathfrak{A}}\}$$

with the same claimed proposal $\mathbb{P}_{Q^A}$ from a set $\mathfrak{Q}^{\mathfrak{A}} \subseteq \mathfrak{R}$ of $|\mathfrak{Q}^{\mathfrak{A}}| = \mathbf{n} - \mathbf{f}$ replicas until timer state t_A expires.

Note that there is no timer for Syncing (ST2) and receiving sufficient SYNC messages is the only way to proceed to Certifying (ST3) of the same view. Some replicas may fall behind due to unreliable communication, failing to receive sufficient SYNC messages while other replicas have reached higher views. To quickly synchronize views, a replica R in view v is allowed to proceed to Syncing (ST2) of view w directly after receiving a set of messages D with views higher than or equal to w :

$$D = \{\text{SYNC}(v', \text{claim}(\mathbb{P}_{Q^D}), \mathbb{CP}_{Q^D}) \mid Q^D \in \mathfrak{Q}^{\mathfrak{D}}\}$$

with $v' \geq w > v$ from a set $\mathfrak{Q}^{\mathfrak{D}} \subseteq \mathfrak{R}$ of $|\mathfrak{Q}^{\mathfrak{D}}| = \mathbf{f} + 1$ replicas. Receiving such $\mathbf{f} + 1$ messages implies that one non-faulty replica has moved to view w after receiving $\mathbf{n} - \mathbf{f}$ SYNC messages



- 1: receive an acceptable proposal
- 2: t_R expires
- 3: receive $\mathbf{n} - \mathbf{f}$ SYNC message of view v
- 4: receive an acceptable proposal
- 5: t_A expires
- 6: receive SYNC messages of view $v' \geq w > v$ from $\mathbf{f} + 1$ replicas

FIGURE 2.4. States in an individual SPOTLESS instance with Rapid View Synchronization

of view $w - 1$, then R can skip to view w directly knowing that at least majority of non-faulty replicas have observed the higher view $w - 1$. To catch up, replica R broadcasts message $S_u = \text{SYNC}(u, \text{claim}(\emptyset), \mathbb{CP}, \Upsilon)$ for each view $u, v \leq u \leq w$, in which Υ is a flag that asks replicas that receive S_u to retransmit the SYNC messages to R they broadcast in view u . With such a design, in SPOTLESS, as long as network remains synchronous, replicas falling behind are capable of catching up actively and immediately, while in previous *rotational* work such as HOTSTUFF, *view synchronization* is assumed by relying on the black-box Pacemaker. In Figure 2.5, we present the pseudo-code of the Rapid View Synchronization part of SPOTLESS. Due to Lemma 2.2.2.1, we have the following:

LEMMA 2.2.4.1. *Assume reliable communication. If replica R conditionally prepares proposal $\mathbb{P}$ by receiving $\mathbf{f} + 1$ SYNC messages with $\text{claim}(\mathbb{P}) \in \mathbb{CP}$, then eventually R will record and conditionally prepare all proposals in $\text{precedes}(\mathbb{P})$.*

PROOF. Let $\mathbb{P}' \in \text{precedes}(\mathbb{P})$. Due to Lemma 2.2.2.1, at least $\mathbf{n} - 2\mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas have sent SYNC messages that claim $\mathbb{P}'$. As such, at-least $\mathbf{f} + 1$ non-faulty replicas will reply to the message $S_u = \text{SYNC}(u, \text{claim}(\emptyset), \mathbb{CP}, \Upsilon)$ sent by replica R . Hence, eventually, R will receive $\mathbf{f} + 1$

Backup role (running at each replica $R \in \mathfrak{R}$ in view v) :

- 1: **event** R enters view v **do**
- 2: state := *recording* (ST1).
- 3: Set timer t_R .
- 4: **event** R receives an acceptable proposal $\mathbb{P}$ **or** t_R expires **do**
- 5: Broadcast SYNC(v , claim($\mathbb{P}$), $\mathbb{CP}$).
- 6: state := *Syncing* (ST2).
- 7: **event** R receives $\mathbf{n} - \mathbf{f}$ SYNC messages of view v **do**
- 8: state := *Certifying* (ST3).
- 9: Sets timer t_A .
- 10: **event** R receives $\mathbf{n} - \mathbf{f}$ SYNC(v , claim($\mathbb{P}$), $\mathbb{CP}$) of the same $\mathbb{P}$ **or** t_A expires **do**
- 11: Enters view $v + 1$.
- 12: **event** R receives $\mathbf{f} + 1$ SYNC messages with views higher than or equal to w , $w > v$ **do**
- 13: Let v be the current view of R .
- 14: **for** each view $u, v \leq u \leq w$ **do**
- 15: Broadcasts SYNC(u , claim($\emptyset$), $\mathbb{CP}$, $\mathfrak{T}$).

FIGURE 2.5. Rapid View Synchronization in instance.

corresponding SYNC messages with claim($\mathbb{P}'$), due to which R will *record* and *conditionally prepare* $\mathbb{P}'$. □

Using Lemma 2.2.4.1, we can prove:

THEOREM 2.2.5. *Let $\mathbb{P}_h$ be the highest proposal that any replica conditionally committed. All non-faulty replicas will eventually record and conditionally prepare all proposals in $\text{precedes}(\mathbb{P}_h)$, when communication becomes synchronous for sufficiently long.*

PROOF. Since $\mathbb{P}_h$ is *locked* by a non-faulty replica, at least $\mathbf{n} - 2\mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas have *conditionally prepared* $\mathbb{P}_h$ (Lemma 2.2.2.1). These $\mathbf{n} - 2\mathbf{f}$ non-faulty replicas either locked $\mathbb{P}_h$ or some proposal with a lower view. Hence, all these replicas will broadcast SYNC messages with claim($\mathbb{P}_h$) $\in \mathbb{CP}$ to convince other replicas to *conditionally prepare* $\mathbb{P}_h$. Hence, when communication becomes synchronous, Lemma 2.2.4.1 completes this proof. □

From Theorem 2.2.5 we know that all non-faulty replicas will learn the same *conditionally committed* chain. However, the replicas may not execute several proposals on the chain until they learn full information of the proposal via the ASK-recovery mechanism detailed in Section 2.2.3.

2.2.5. Mechanism Guaranteeing Liveness. In some cases, replica R cannot make any progress unless it receives some specific messages from other replicas:

- (1) R cannot switch from Syncing (ST2) to Certifying (ST3) unless it receives $\mathbf{n} - \mathbf{f}$ SYNC messages of its current view.
- (2) R cannot catch up to learn a path from a *conditionally prepared* proposal to the genesis proposal unless at least $\mathbf{f} + 1$ other replicas reply to its SYNC messages with flag Υ , which requires the receivers to retransmit the SYNC messages that they broadcast before.
- (3) R cannot *record* a proposal it did not receive from the leader, unless any replica replies to its ASK message by forwarding the corresponding PROPOSE message.

However, due to unreliable communication, R may fail to receive messages, e.g., replies to SYNC messages with flag Υ or replies to ASK messages. To deal with this case, R will periodically retransmit the messages until it receives the necessary replies.

In an asynchronous environment, one cannot reliably distinguish between communication failure (e.g., due to long and unpredictable message delays) and replica failure. Hence, consensus protocols such as HOTSTUFF [164] and many others [42, 108] simply assume to be operating after a *Global Synchronization Time*, at which point all communication is bound by some message delay such that all replicas can always reliably determine in which view they operate [164]. Such a design is inflexible in the presence of true asynchronous communication, however.

Instead, SPOTLESS instances use our *Rapid View Synchronization* mechanism to allow replicas to figure out in which view they should operate. To adapt to fluctuations in message delays, SPOTLESS will adjust the timeout interval used by individual replicas to detect replica failures. As message delays in typical deployments do not often change drastically, we choose to *not* use a traditional exponential backoff mechanism [76], but instead to adjust the timeout interval of replicas R in a more moderate way. For consecutive timeouts of the same timer in consecutive views, we only increase the timeout interval by a constant ε (after each consecutive view). If a replica receives an expected message for which the timeout interval was Δ *before* 0.5Δ , then the replica reduces the timeout by half. We have the following technical result.

LEMMA 2.2.5.1. *Let v be the highest view reached by a non-faulty replica after communication enters a period of synchronous communication. Non-faulty leader L_w with $v + 2 \leq w$ is capable of*

finding a proposal $\mathbb{P}'$ such that all non-faulty replicas will accept a proposal $\mathbb{P}$ extending from $\mathbb{P}'$, $\mathbb{P} := \text{PROPOSE}(w, \tau, \text{claim}(\mathbb{P}'))$.

PROOF. Let $\mathbb{P}'$ be the highest proposal *locked* by any non-faulty replicas after view $w - 1$ (in the worst case, $\mathbb{P}'$ is the genesis proposal). As $\mathbb{P}'$ was locked by a non-faulty replica, at least $\mathbf{n} - 2\mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas have *conditionally prepared* $\mathbb{P}'$ before view $w - 1$. For any replica R among these $\mathbf{n} - 2\mathbf{f}$ replicas, either R *locked* on $\mathbb{P}'$ or $\mathbb{P}'$ is from a newer view than the locked proposal of R .

Hence, in view $w - 1$, the SYNC messages of the $\mathbf{n} - 2\mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas would include $\mathbb{P}'$ in $\mathbb{CP}$. As such, all non-faulty replicas would *conditionally prepare* $\mathbb{P}'$ and inform other replicas, and leader L_w would learn that at least $\mathbf{n} - 2\mathbf{f}$ non-faulty replicas have *conditionally prepared* $\mathbb{P}'$ before entering view w . As such, leader L_w can propose a new proposal $\mathbb{P} := \text{PROPOSE}(w, \tau, \text{claim}(\mathbb{P}^*))$, with $\mathbb{P}^* = \mathbb{P}'$ or with $\mathbb{P}^*$ being some proposal with a higher view than $\mathbb{P}'$ (if leader L_w knows such an *extendable* higher proposal). All non-faulty replicas will consider $\mathbb{P}$ as *acceptable* since their *locked* proposal is either $\mathbb{P}'$ or a proposal from an lower view. $\square$

As a direct consequence, we have the following corollary.

COROLLARY 2.2.1. *Let v be the highest view reached by a non-faulty replica after communication enters a period of synchronous communication. If all non-faulty replicas have internal timers that are higher than the current maximum message delay, then the proposals of any non-faulty leader L_w during view w , $w \geq v + 2$, will be conditionally prepared after view w by all non-faulty replicas.*

PROOF. We denote to be t the time that L_w enters Certifying (3) of view w . From Lemma 2.2.5.1 we know that all non-faulty replicas will *accept*. By t , at least $\mathbf{f} + 1$ non-faulty replicas have sent $\text{SYNC}(w, \text{claim}(m), \mathbb{CP})$. By $t + \delta$, non-faulty replicas falling behind will have received at least $\mathbf{f} + 1$ such SYNC messages, entered view w and voted for m . By $t + 2\delta$, all non-faulty replicas will have received at least $\mathbf{n} - \mathbf{f}$ such SYNC messages and *conditionally prepare* m . $\square$

Corollary 2.2.1 is at the basis of proving termination: consensus decisions are eventually made when communication is synchronous.

THEOREM 2.2.6. *All non-faulty replicas will eventually commit new proposals after communication enters a sufficiently-long period of synchronous communication.*

PROOF. First, we note that all non-faulty replicas will eventually satisfy the internal timer requirements, as they will increase their internal timer until they can successfully participate in the replication protocol. Hence, by Corollary 2.2.1 a non-faulty leader L_w of view w will eventually be able to propose a message $\mathbb{P}$ that will be conditionally prepared. As there are $\mathbf{n} > 3\mathbf{f}$ replicas, after view w , there will eventually be a sequence of three consecutive non-faulty leaders. These leaders will be able to make proposals satisfying the conditions of Definition 2.2.2 to assure a proposal will get committed. $\square$

2.3. Concurrent Consensus

The main benefit of rotational consensus, as used by SPOTLESS and HOTSTUFF [164], is that a single proposal represents the entire chain of preceding proposals. This greatly reduces the message complexity of view-changes when compared to traditional stable-leader consensus protocols such as PBFT [40].

Unfortunately, rotational consensus requires that consecutive consensus decisions are made one-at-a-time, thereby preventing the usage of *out-of-order processing* to maximize throughput. This makes HOTSTUFF and individual rotational consensus instances of SPOTLESS significantly slower than traditional consensus protocols such as PBFT in practical deployments: leaders in PBFT can use *out-of-order processing* to propose client requests for future views while waiting on the current consensus round to finish, thereby maximizing the utilization of the network bandwidth available at the leader independent of any message delays (which dominate the time it takes to finish a single consensus round).

As an alternative to out-of-order processing, SPOTLESS will adopt concurrent consensus [77, 151]. By running multiple concurrent instances, SPOTLESS is able to effectively utilize all network bandwidth and computational resources available: when one SPOTLESS instance is waiting for a proposal to be processed (e.g., waiting for SYNC messages), other SPOTLESS instances use the available network bandwidth and computational resources to propose additional requests.

2.3.1. Concurrent Instances in SpotLess. In SPOTLESS, the system runs $\mathbf{m}$, $1 \leq \mathbf{m} \leq \mathbf{n}$, SPOTLESS instances concurrently. Instances do not interfere with each other. Each instance

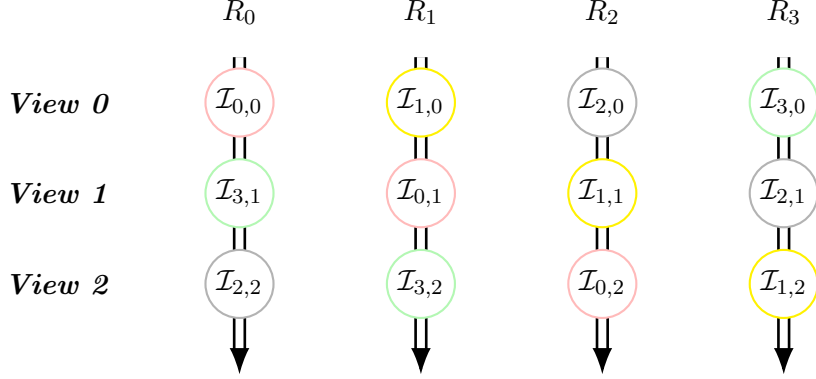


FIGURE 2.6. Primary rotation in SPOTLESS with four replicas and four instances. The circles in the form of $\mathbb{P}_{i,v}$ on the arrow of R_r represent that R_r is the leader of instance $\mathcal{I}_i$ in view v , where $r = (i + v) \bmod \mathbf{n}$.

independently deals with any malicious behavior. To enforce that each instance is coordinated by a distinct leader, the leader of $\mathcal{I}_i$ in view v is predetermined: $\text{id}(L_{i,v}) = (i + v) \bmod \mathbf{n}$. Figure 2.6 shows how SPOTLESS assigns and rotates leader in each SPOTLESS instance. As all instances rotate over all leaders, all instances are equally affected by malicious behavior. When given the choice, non-faulty replicas will prioritize instances that are in older views over other instances. Due to leader rotation and this instance prioritization, the view of all instances will remain roughly-in-sync.

Each SPOTLESS instance determines a local order of proposals. All committed proposals on the chains are totally ordered and then executed. We order the committed proposals among different instances by their view and instance identifier. We order all proposals from low view to high view and from instance $\mathcal{I}_0$ to instance $\mathcal{I}_{\mathbf{m}-1}$. Figure 2.7 shows the *total ordering* in SPOTLESS. Finally, each replica *executes* the committed proposals in order and informs the clients of the outcome of their requested transactions.

As individual SPOTLESS instances provide consensus and combining consensus decisions of instances is deterministic, we conclude

THEOREM 2.3.1. *All instances of SPOTLESS will eventually commit new proposals after communication enters a sufficiently-long period of synchronous communication.*

2.3.2. Benefits of Concurrent Processing. Theoretically, if we run $\mathbf{m}$, $1 \leq \mathbf{m} \leq \mathbf{n}$ instances concurrently, SPOTLESS could achieve $\mathbf{m}$ times the throughput of a single instance. As

the number of replicas $\mathbf{n}$ and instances $\mathbf{m}$ scales, the throughput of SPOTLESS will keep growing until the throughput reaches resource bottlenecks (e.g., limited computational power or network bandwidth). Scaling beyond these resource bottlenecks, the throughput of SPOTLESS decreases as the system scales further due to the added communication cost to reach consensus among more replicas.

Next, we shall theoretically model the benefits of *concurrent consensus* in SPOTLESS. Assume the message delay is Δ and the network bandwidth is B , and that proposal includes a *batch* of β individual transactions. Hence, the best-case throughput of individual SPOTLESS instances with $\mathbf{n}$ replicas is

$$T_{\text{SPOTLESS}_1} = \frac{\beta}{t_{\text{leader}} + 2\Delta}; \quad t_{\text{leader}} = \frac{S_{\text{leader}}}{B}$$

In the above, S_{leader} is the size of all $\mathbf{n} - \mathbf{f}$ SYNC messages the leader receives and of all $\mathbf{n} - \mathbf{f}$ PROPOSE messages the leader sends and t_{leader} is the time the leader is busy sending and receiving.

If network bandwidth is not the bottleneck, then the maximum performance of $\mathbf{n}$ SPOTLESS instances is $\mathbf{n}T_{\text{SPOTLESS}_1}$. If we run $\mathbf{n}$ SPOTLESS instances concurrently, one for each replica, the communication complexity per instance is $O(\mathbf{n}^2)$ in each round. Thus, as the system reaches resource bottlenecks, adding more instances will cause no gains in throughput. If we assume that network bandwidth is the bottleneck, the best-case throughput of SPOTLESS with $\mathbf{n}$ instances and $\mathbf{n}$ replicas is upper-bounded by

$$T_{\text{SPOTLESS}_{\text{bw}}} = \frac{\mathbf{n}B\beta}{S_{\text{leader}} + (\mathbf{n} - 1)S_{\text{backup}}}$$

In the above, S_{backup} is the size of all messages a replica sends and receives as part of the backup role in one view, and $S_{\text{leader}} + (\mathbf{n} - 1)S_{\text{backup}}$ is the sum of all bandwidth usage by a single replica during $\mathbf{n}$ instances of a single view. Due to the streamlined design of SPOTLESS, SPOTLESS instances end up sending fewer messages per consensus decision than PBFT. Consequently, when scaling up to many replicas (and many instances), the best-case throughput of SPOTLESS will be higher than concurrent consensus based on PBFT (e.g., RCC [77]).

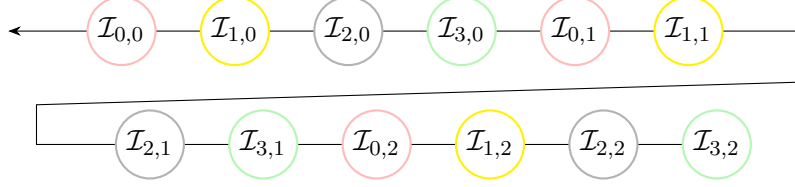


FIGURE 2.7. Total ordering of the twelve proposals of Figure 2.6 made among four concurrent instances in three consecutive views.

2.4. Clients and Transaction Execution

Lastly, we discuss how SPOTLESS provides service to clients. Observe that to maximize throughput, it is beneficial to assure that no two instances propose the same transactions. In addition, it is important to balance all requests over all instances. Furthermore, we must also consider malicious leader behavior aimed at refusing service to some non-faulty clients.

Unlike systems such as RCC [77], which initially assigns every client to a single leader (that coordinates a single instance), SPOTLESS assigns every client request to a single instance based on its digest: instance $\mathcal{I}_i$, $1 \leq i \leq \mathbf{m}$, can only propose transactions with digest d such that $(i - 1) = d \bmod \mathbf{m}$. Hence, multiple requests by the same client will be handled by different instances. Since we adopt a cryptographically strong hash algorithm to compute digests, this allocation of transactions to instances also ensures load balance among instances.

When a non-faulty replica commits a transaction, it will execute that transaction in an order consistent with the total ordering of all transactions (across all instances). After executing a transaction, the replica informs the client of the result via an INFORM message. A client c randomly sends a transaction τ to a replica R_j , starts a timer t_C , and awaits for $\mathbf{f} + 1$ identical INFORM responses. If c fails to get these $\mathbf{f} + 1$ responses before t_C expires, it sends the transaction to the next replica R_{j+1} and doubles the timeout interval. The client keeps doing so until it receive $\mathbf{f} + 1$ identical INFORM responses confirming that the transaction has been executed. Due to leader rotation, *every* replica will eventually be the leader of an instance that can propose τ , including a non-faulty replicas that will eventually propose τ .

Note that transaction execution in view v requires that all instances finished view v (hence, we know the total ordering of the transactions successfully proposed in view v and that need to be executed). In cases where system load is low, an instance leader L can end up not receiving any client transactions while other leaders are already proposing transactions for their instances. To

prevent that executing of the proposals of other instances has to wait until L is able to propose a transaction, leader L can propose a no-op transaction if no other transactions are available. As we already proved *termination* (Theorem 2.3.1) and *non-divergence* (Theorem 2.2.3) and the mechanisms outlined above guarantee *service*, we finally conclude

THEOREM 2.4.1. SPOTLESS *provides consensus*.

2.5. Evaluation

Previously, we detailed and analyzed the design of SPOTLESS, showing several theoretical advantages when compared to its peers. Next, to show the practical advantages of SPOTLESS, we will experimentally evaluate its performance, both in the normal case and during Byzantine failures. In our evaluation, we compare the performance of SPOTLESS in APACHE RESILIENTDB (Incubating), our high-performance open-source blockchain database, with the well-known leader-based consensus protocols PBFT [40], HOTSTUFF [164], NARWHAL-HS [49], and our PBFT-based concurrent consensus paradigm RCC [77]. We focus on answering the following questions:

- Q1 Scalability: does SPOTLESS deliver on the promises to provide better scalability than other consensus protocols?
- Q2 Latency: does SPOTLESS provide low client latency while providing high throughput?
What factors affect latency and throughput?
- Q3 What is the impact of batching client transactions on the performance of SPOTLESS?
- Q4 How does SPOTLESS perform in presence of Byzantine failures?
- Q5 How does *concurrent consensus* improve performance?

To study the practical performance of SPOTLESS and other consensus protocols, we implemented SPOTLESS and other protocols in APACHE RESILIENTDB. To generate experimental workloads, we used the *Yahoo Cloud Serving Benchmark* [46] provided by the Blockbench macro benchmarks [54]. In the generated workload, each client transaction queries a YCSB table with half a million active records and 90% of the transactions write and modify records. Before the experiments, each replica is initialized with an identical copy of the YCSB table. We perform all experiments on Oracle Cloud, using up to 128 machines for replicas and 32 machines for clients. Each replica and client is deployed on a e3-machine with a 16-core AMD EPYC 7742 processor, running at 3.4 GHz, and with 32 GB memory.

2.5.1. The APACHE RESILIENTDB Blockchain Database. The architecture of APACHE RESILIENTDB is heavily *multi-threaded* and *pipelined* and is optimized for maximizing throughput [71, 72, 73, 84, 135]. To further maximize throughput and minimize the overhead of any consensus protocol, APACHE RESILIENTDB has built-in support for *batching* client transactions. We typically group 100 txn/batch. In this case, the size of a proposal is 5400 B and of a client reply (for 100 transactions) is 1748 B. The other messages exchanged between replicas during the replication algorithm have a size of 432 B.

To maximize performance of PBFT-like protocols, APACHE RESILIENTDB supports *out-of-order processing* of transactions (see Section 2.3) in which leaders can propose future transactions before current transactions are executed. The streamlined designs of SPOTLESS and HOTSTUFF cannot utilize out-of-order processing, however.

In APACHE RESILIENTDB, each replica maintains an immutable *blockchain ledger* that holds an ordered copy of all executed transactions. The ledger not only stores all transactions, but also strong cryptographic proofs of their acceptance by a consensus protocol. This ledger can be used to provide *strong data provenance*.

In our experiments, replicas not only perform consensus but also communicate with clients and execute transactions. In this practical setting, performance is not fully determined by the cost of consensus, but also by the cost of *communicating with clients*, of sequential *execution* of all committed transactions, of *cryptography*, and of other steps involved in processing messages and transactions. The *sequential execution* bottleneck of APACHE RESILIENTDB in our deployments, the maximum speed by which APACHE RESILIENTDB can execute transactions without performing any other tasks, is 340 ktxn/sec.

To reduce the overhead of sending individual messages, APACHE RESILIENTDB uses *message buffering*: APACHE RESILIENTDB collects messages intended for specific receiver in a buffer and sends all messages at-once when the buffer reaches a threshold value. For each consensus protocol we study, the threshold value used has been configured to maximize throughput.

In addition, to better utilize the computing and network resources, APACHE RESILIENTDB optimizes SPOTLESS in the following two ways. First, leaders broadcast the actual content of client requests in advance, and during the proposal step, they only operate on the digests of the requests. Second, in the geo-scale experiments that will be discussed later, leaders can broadcast a new

proposal optimistically in a fast path before receiving $2f + 1$ votes for the proposal in the previous view. If any Byzantine behavior is detected, replicas switch back to the slow original path the next time they become the rotated leader.

2.5.2. The Consensus Protocols. We evaluate the performance of SPOTLESS by comparing it with a representative selection of four efficient practical consensus protocols implemented in APACHE RESILIENTDB:

PBFT [40]. We use heavily optimized out-of-order implementation that uses message authentication codes.

RCC [77]. RCC turns PBFT into a concurrent consensus protocol.

HOTSTUFF [164]. HOTSTUFF uses threshold signatures to minimize communication. Since existing threshold signature algorithms are expensive and quickly become the bottleneck, we use a list of $n - f$ `secp256k1` digital signatures to represent a threshold signature, which improves our throughput. In our experiments, we implement the pipelined STREAMLINED HOTSTUFF.

NARWHAL-HS [49]. NARWHAL-HS separates the replication of transactions and ordering transactions, enabling concurrent transaction dissemination. We simulate the communication complexity and computational overhead of NARWHAL-HS by running HOTSTUFF and requiring replicas to broadcast messages consisting of a client batch and $2f + 1$ digital signatures.

We run the concurrent protocols RCC and SPOTLESS with n instances unless stated otherwise.

2.5.3. Experiments. To be able to answer Questions Q1–Q5, we perform fifteen experiments in which we measure the performance of SPOTLESS and other consensus protocols. We measure *throughput* as the number of transactions that are executed per second and *latency* as the average duration between when the client sends a transaction and when the client receives $f + 1$ corresponding responses. Unless stated otherwise, all replicas are non-faulty. We run each experiment for 130 s (except the seventh experiment): the first 10 s are warm-up, and measurement results are collected over the next 120 s. We average our results over three runs.

In the *scalability experiment*, we measure throughput as a function of the number of replicas. We vary the number of replicas between $n = 4$ and $n = 128$ and we use a batch size of 100 txn/batch. The results can be found in Figure 2.8(a).

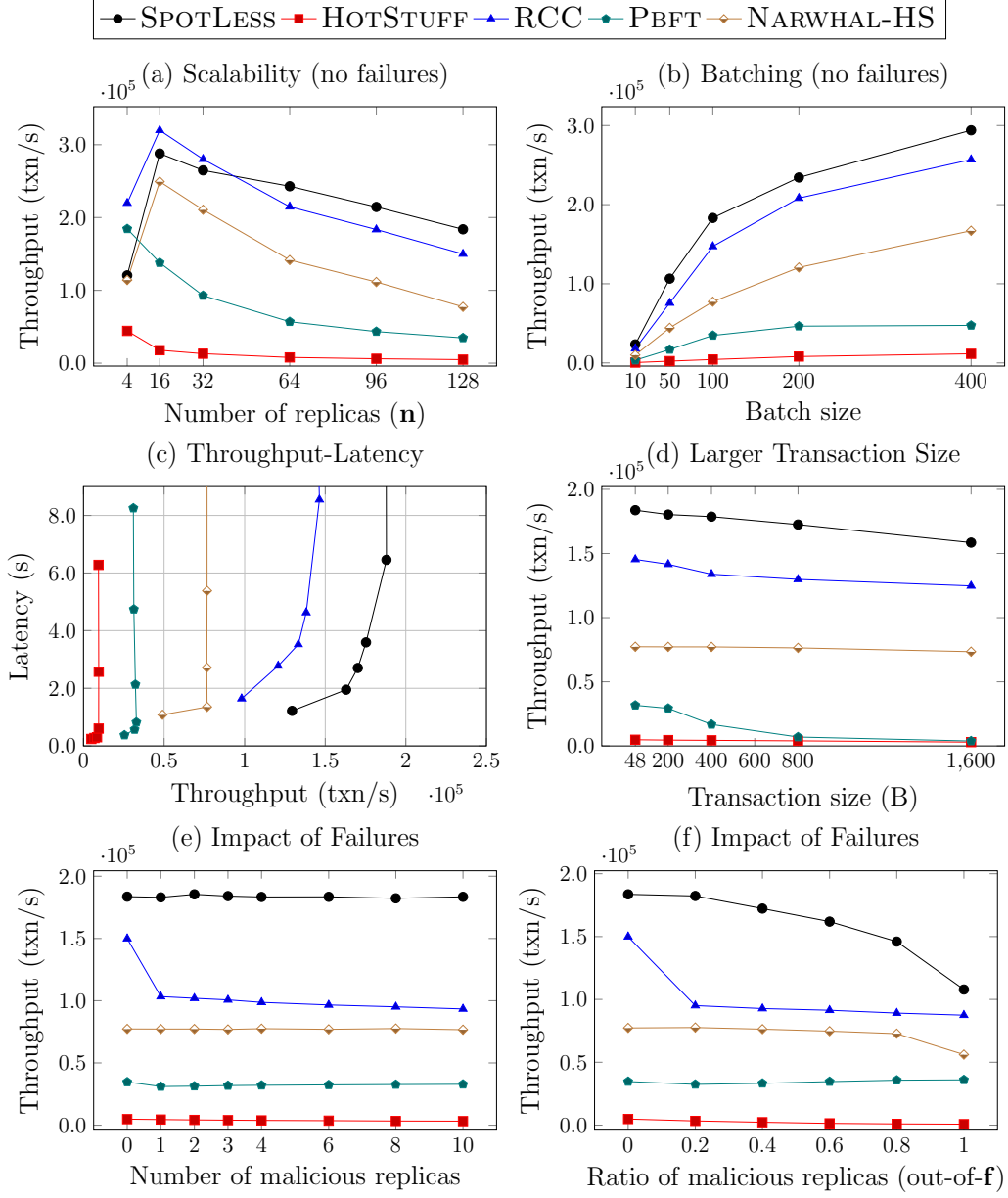


FIGURE 2.8. Performance of SPOTLESS and other protocols. The number of replicas is 128 except in (a).

In the *batching experiment*, we measure the throughput as a function of the number of replicas. We use $n = 128$ replicas and we vary batch size between 10 txn/batch and 400 txn/batch. The results can be found in Figure 2.8(b).

In the *throughput-latency* experiment, we measure the latency as a function of the throughput. We use $n = 128$ replicas, set the batch size to 100 txn/batch, and we vary the speed by which

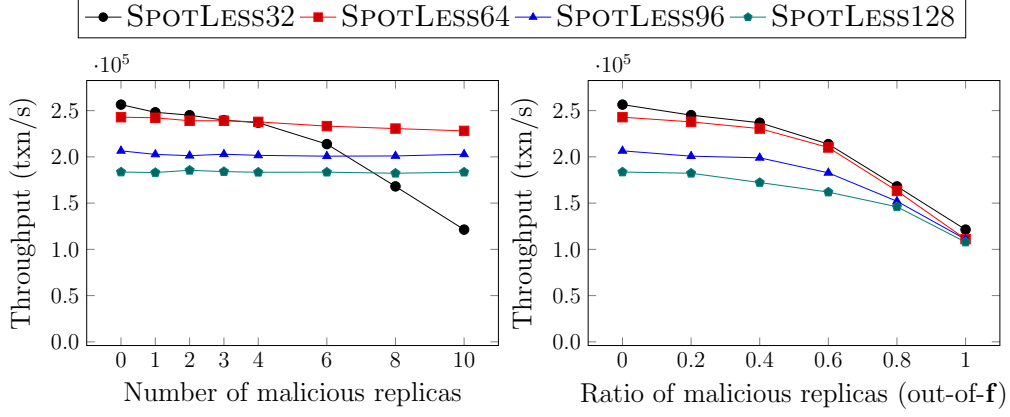


FIGURE 2.9. Performance of SPOTLESS during failures as a function of the number of replicas and faulty replicas.

each leader receives client requests to affect throughput and latency. The results can be found in Figure 2.8(c).

In the *transaction-size* experiment, we measure the throughput of SPOTLESS as a function of the individual YCSB transaction size. We use $n = 128$ replicas and vary the transaction size from 48 B to 1600 B. The results can be found in Figure 2.8(d).

In the *all-throughput-failures* experiment, we measure the throughput as a function of the number of malicious replicas that do not participate in consensus. We use $n = 128$ replicas, and we vary the number of faulty replicas between 0 and 10 or between 0 and f . We make the faulty replicas non-responsive at the same time point and measure throughput afterward for 120 s. We set the timeout length in SPOTLESS, HOTSTUFF, and NARWHAL-HS based on the calculated average view duration and set the timeout length in RCC and PBFT based on the average client latency. The results can be found in Figure 2.8(e) and (f).

In the *SPOTLESS-throughput-failures* experiment, we further measure the throughput of SPOTLESS as a function of the number of replicas and the number of malicious replicas that do not participate in consensus. We vary the number of replicas between $n \in \{32, 64, 96, 128\}$ and we perform two measurements for each n . We vary the number of faulty replicas between 0 and 10 or between 0 and f . Based on the calculated average view duration, we have set the timeout length in SPOTLESS appropriately. The results can be found in Figure 2.9.

In the *throughput-latency-failure experiment*, we measure the latency of SPOTLESS and RCC as a function of throughput in the presence of malicious replicas. We use $n = 128$ replicas and set

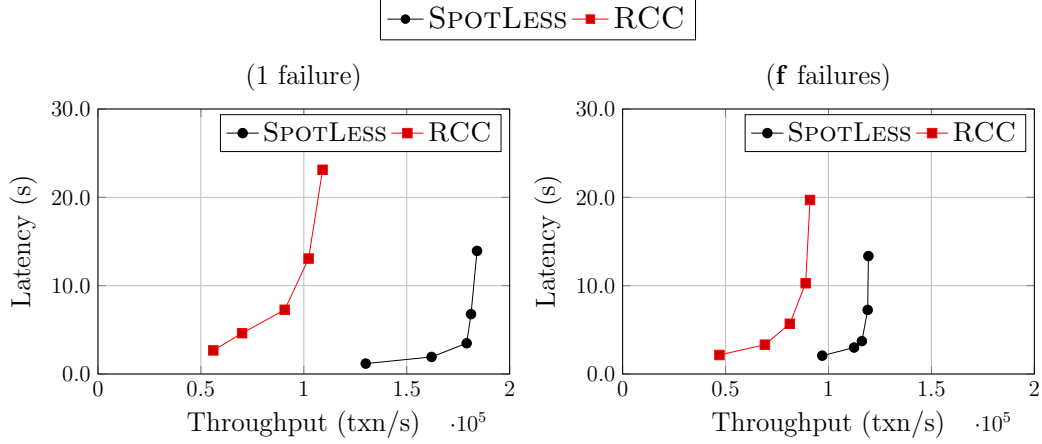


FIGURE 2.10. System throughput-latency of SPOTLESS and RCC in the presence of failures (128 replicas).

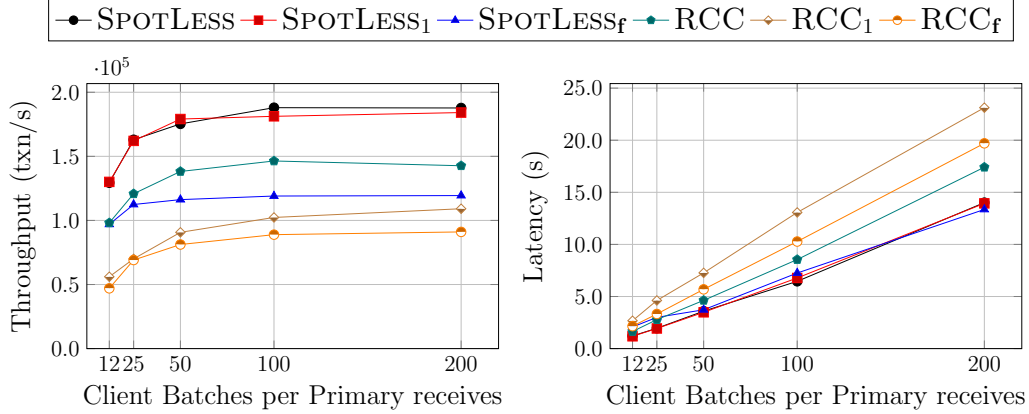


FIGURE 2.11. Throughput and latency of SPOTLESS and RCC as a function of the amount of parallel transaction processing (128 replicas).

the number of faulty, non-responsive, replicas to be 1 or f . We vary the number of client batches that each leader receives in the same way as described in the throughput-latency experiment. We only count the latency of proposals that are sent to non-faulty replicas. The results can be found in Figure 2.10.

In the *parallel transaction processing experiment*, we measure the performance of SPOTLESS and RCC as a function of the amount of *concurrent* (both protocols) and *out-of-order* (RCC) processing. To do so, we measure the throughput and latency of SPOTLESS and RCC as a function of the number of client batches that each leader receives. We use $n = 128$ replicas and we vary the number of client batches between 12 and 200. The results can be found in Figure 2.11.

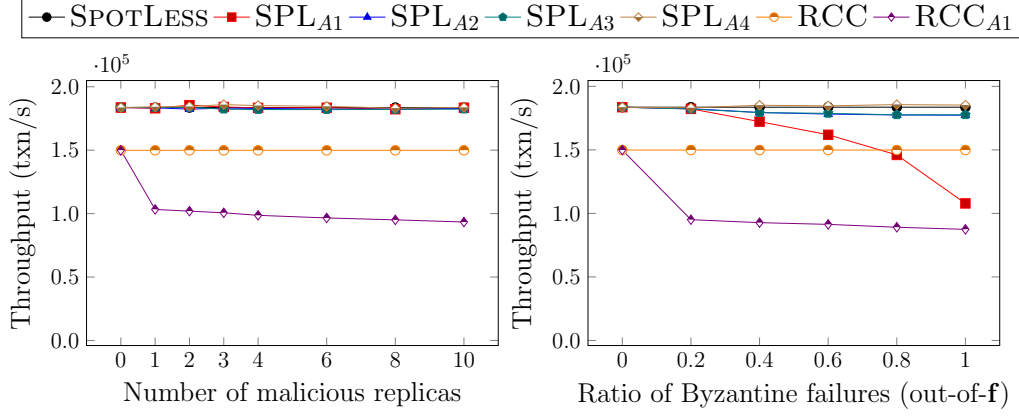


FIGURE 2.12. Performance of SPOTLESS and RCC during Byzantine failures (128 replicas, four attack scenarios for SPOTLESS). *Note: SPL is the abbreviation for SPOTLESS here.*

In the *throughput-Byzantine* experiment, we measure the throughput of SPOTLESS in the presence of attacks as a function of the number of Byzantine replicas. We consider four types of attacks:

- A1 faulty replicas are non-responsive;
- A2 faulty replicas act *malicious* when they are the leader by keeping $\mathbf{f}$ non-faulty replicas *in the dark* (by not sending proposals to them);
- A3 faulty replicas act *malicious* by sending conflicting concurring votes in an attempt to cause *divergence*: they send one message to $\mathbf{f}$ non-faulty replicas and a different one to the other non-faulty replicas; and
- A4 faulty replicas act *malicious* by refusing to participate in the consensus of proposals from non-faulty leaders, this in an attempt to subvert non-faulty leaders (and make them look faulty).

For comparison, we include RCC. The *victims* of these attacks are the replicas that are kept in the dark (A2), receive a proposal that is received by not more than $\mathbf{f}$ non-faulty replicas (A3), or are not responded to (A4). The throughput of RCC is not influenced by A2, A3, and A4 as long as the number of *victims* is not greater than $\mathbf{f}$. Hence, we only include the normal-case (failure-free) throughput of RCC and the throughput during A1 for comparison. We use $\mathbf{n} = 128$ replicas and we vary the number of malicious replicas between 0 and 10 or between 0 and $\mathbf{f}$. The results can be found in Figure 2.12.

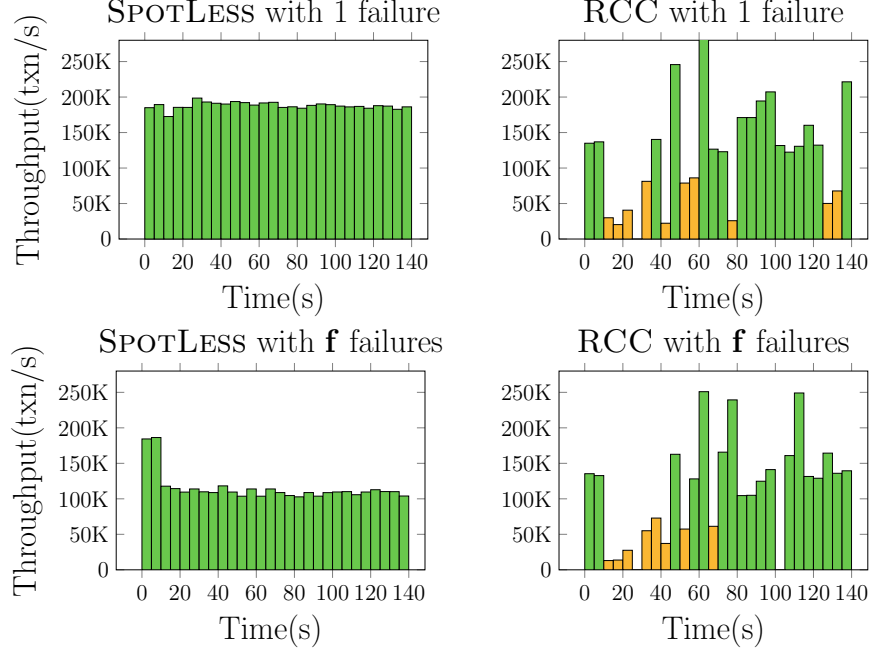


FIGURE 2.13. Throughput timeline of SPOTLESS and RCC after injecting failures.

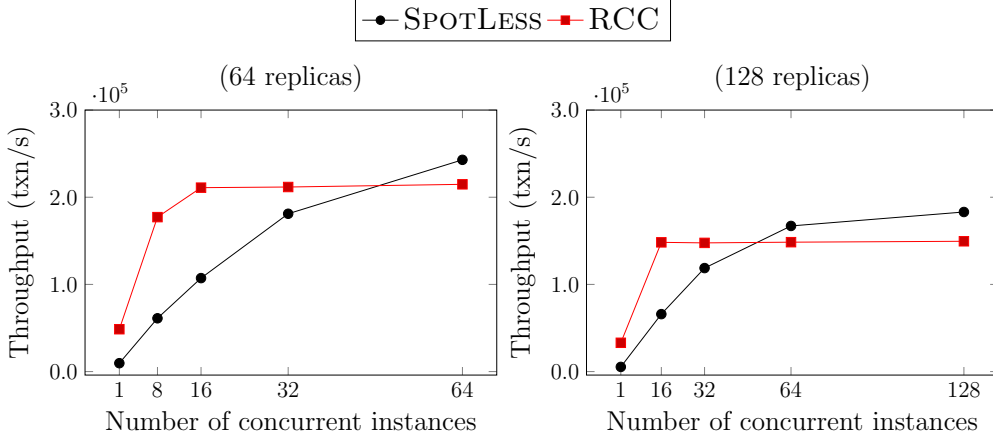


FIGURE 2.14. Performance of SPOTLESS and RCC as a function of the number of concurrent instances.

In the *real-time-throughput-failure* experiment, we measure the real-time throughput of SPOTLESS and RCC after making malicious replicas non-responsive as a function of time. We use $n = 128$ replicas, and we set the number of faulty replicas to 1 or f . We run the experiments for 140s, record throughput every 5 seconds, and have the failures happen at the 10th second. The results can be found in Figure 2.13.

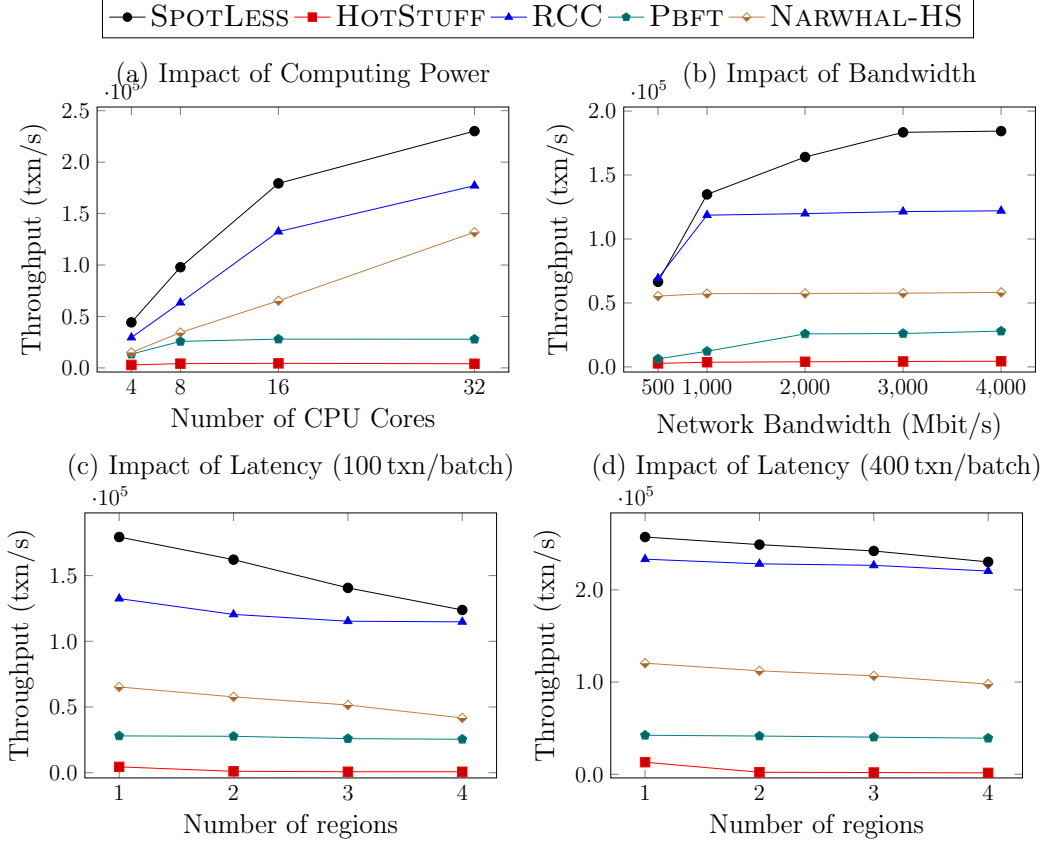


FIGURE 2.15. Impact of computing and network resources on the performance with 128 replicas.

In the *concurrent-consensus* experiment, we measure the throughput of SPOTLESS and RCC as a function of the number of replicas and concurrent instances. We use $n \in \{32, 128\}$ replicas and we vary the number of concurrent instances between 1 and n . The results can be found in Figure 2.14.

In the *computing-power-impact* experiment, we measure the throughput of SPOTLESS and other protocols as a function of the number of CPU cores in each replica. We use $n = 128$ replicas and we vary the number of CPU cores between 4 and 32. The results can be found in Figure 2.15(a).

In the *network-bandwidth-impact* experiment, we measure the throughput of SPOTLESS and other protocols as a function of the bandwidth. We use $n = 128$ replicas and vary the bandwidth between 500 Mbit/s and 4000 Mbit/s using FireQOS [158], a program that helps configure traffic shaping on Linux. The results can be found in Figure 2.15(b).

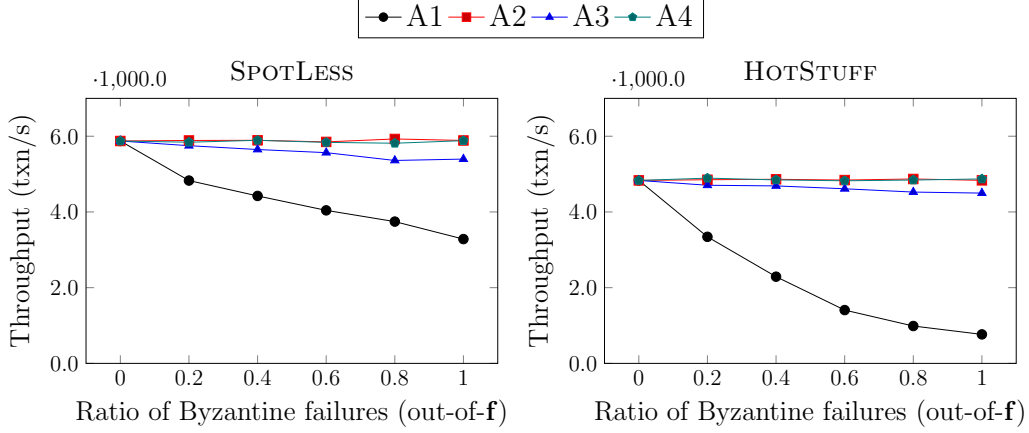


FIGURE 2.16. Performance of single-instance SPOTLESS and HOTSTUFF with failures.

In the *global-regions* experiment, we measure the throughput of SPOTLESS and other protocols as a function of the number of regions. We use $n = 128$ replicas and vary the number of regions between 1 and 4. For each run, the 128 replicas are uniformly distributed in the regions Oregon, North Virginia, London, and Zurich. The results can be found in Figure 2.15(c) and (d).

In the *non-concurrent-failure* experiment, we measure the throughput of single-instance SPOTLESS and HOTSTUFF as a function of the number of faulty replicas. We use $n = 128$ replicas and vary the number of malicious replicas between 0 and f . The results can be found in Figure 2.16.

2.5.4. Experiment Analysis. The experimental results presented in the previous sections allows us to answer the research questions Q1–Q5. First, we observe that increasing the batch size, increases the performance of all consensus protocols, thereby answering Q3 as expected. Since the gains brought by increased the batch size are small after 100 txn/batch, we used 100 txn/batch in all other experiments.

SPOTLESS outperforms all other protocols in failure-free conditions (Q1, Q2). As Figure 2.1 shows, the amortized message complexity per decision is n^2 for SPOTLESS, while it is $2n^2$ for RCC. Hence, as SPOTLESS has fewer messages to process, SPOTLESS can even outperform RCC by up to 23%. Due to the *message buffer mechanism*, SPOTLESS and RCC require sufficient batches of client requests to fill the system pipeline. Otherwise, the two protocols may get stalled since no messages are sent and processed. When the pipeline is full, the latency of both SPOTLESS and RCC in APACHE RESILIENTDB is dominated by the maximum throughput, as Figure 2.8(c),

2.10, and 2.11 show. The higher throughput is, the shorter a client request waits to be proposed and then the lower latency is. Thus, even though SPOTLESS needs more communication phases than RCC to commit a proposal, SPOTLESS has a lower latency by up to 32% than RCC. Also, SPOTLESS outperforms NARWHAL-HS because, for each committed block, SPOTLESS verifies $O(n)$ MACs while NARWHAL-HS verifies $O(n)$ digital signatures.

By introducing concurrent processing, SPOTLESS is able to outperform HOTSTUFF, the other streamlined rotational consensus protocol, by up to 3803%. In this situation, the performance of HOTSTUFF is bottlenecked by the message delay due to the lack of out-of-order processing. From Figure 2.8(d) we conclude that RCC and SPOTLESS are able to sustain high throughput even if we increase the transaction size to 1600 B per YCSB transaction, whereas the throughput of PBFT and HOTSTUFF decreases greatly. This is easily explained: the concurrent design of RCC and SPOTLESS balances the leader’s load to all replicas, whereas in PBFT and HOTSTUFF the performance is bottlenecked by the bandwidth available to the single proposing leader.

As Figure 2.9 shows, non-responsive faulty replicas negatively affect the performance of SPOTLESS in all cases (Q4): indeed, non-responsive faulty replicas do not perform their leader role, due to which the non-faulty replicas can only wait until their timers expire to switch out these faulty leaders. The larger the number of replicas, the smaller the relative influence of faulty replicas on performance. For example, when there are f faulty replicas, the throughput of SPOTLESS128 decreases by 41% while that of SPOTLESS32 decreases by 54%. Due to *concurrent consensus*, a larger number of replicas implies more SPOTLESS instances with non-faulty leaders that utilize CPU resources while waiting for the instances with non-responsive leaders.

From Figure 2.8, 2.10, and 2.11, we also know that SPOTLESS shows great resilience to non-responsive faulty replicas when compared with other protocols (Q4). In a deployment with 128 replicas, the first 120s after failures happen, SPOTLESS shows a gain in throughput over other protocols and a lower latency (Q1, Q2) despite the number of faulty replicas.

Thanks to the *ASK-recovery mechanism* and *Rapid View Synchronization*, described in Section 2.2, SPOTLESS shows strong resilience to Byzantine attacks, as we can observe from the results in Figure 2.12. No matter the type of attack, the *victims* can quickly detect the failure and catch up by receiving $f + 1$ SYNC messages from other non-faulty replicas and sending ASK messages.

When facing non-responsive faulty replicas, the two mechanisms are useless, however, as timing out instances is the only way to advance view in this case.

The results of Figure 2.13 show obvious fluctuations in the real-time throughput of RCC after injecting failures. This is due to the usage of an exponential back-off penalty algorithm to ignore instances with faulty leaders. Eventually, the throughput of RCC gradually recovers to the original level and then keeps stable. This is the best case for RCC, however, as all f failures happen at the same time. If the failures appear one by one, RCC will suffer from these low-throughput fluctuations (the yellow columns in Figure 2.13) during each failure. In contrast, SPOTLESS presents a more stable throughput timeline after failures happen (Q4).

Figure 2.14 shows that SPOTLESS benefits more from concurrent consensus than RCC, especially when there are many instances. (Q5). When there are 32 or fewer instances, RCC outperforms SPOTLESS because RCC enables *out-of-order processing* in individual instances, which is not supported by the streamlined design of SPOTLESS instances. As the number of concurrent instances increases, the throughput of RCC reaches a message processing bottleneck when there are 16 instances and then remains stable, whereas the throughput of SPOTLESS can increase further due to the lower message complexity and reaches its peak value when there are n instances, higher than RCC by up to 23%.

The performance of consensus protocols is significantly influenced by computing and network resources. First, Figure 2.15(a) shows that the performance of all protocols decreases when compute power is restricted (fewer CPU cores in each replica). Second, Figure 2.15(b) shows that decreasing the network bandwidth negatively impacts the performance of all protocols in which network is a bottleneck. We note that NARWHAL-HS is barely affected, however, as it is limited by computing resources as it has to verify $n - f$ digital signatures per block. Similarly, Figure 2.15(c) and (d), show that *increasing* the number of regions, which not only decreases network bandwidth but also increases latency, *negatively impacts* the performance of all protocols. We notice that in all cases, SPOTLESS maintains a higher performance than RCC, even though SPOTLESS is affected by decreasing bandwidth. Finally, the comparison between Figure 2.15(c) and (d), shows that that increasing the batch size can partially mitigate bandwidth bottlenecks.

Figure 2.16 shows that the presence of Byzantine failures has similar negative effects on the performance of single-instance SPOTLESS and HOTSTUFF. In all cases, the throughput of single-instance SPOTLESS is higher than that of HOTSTUFF due to the lower computation costs of verifying signatures in SPOTLESS (as compared to dealing with the threshold signatures used in HOTSTUFF). Hence, compared with HOTSTUFF, replicas in SPOTLESS are able to respond more quickly, lowering the per-round latency and increasing throughput.

Based on our findings, we conclude that SPOTLESS makes full use of *concurrent consensus* (Q5), provides higher throughput and better scalability than any other consensus protocol (Q1), and does so with low latency (Q2). Moreover, client batching does benefit the performance of SPOTLESS (Q3). Finally, we conclude that SPOTLESS can efficiently deal with failures (Q4) thanks to *concurrent consensus*, the *ASK-recovery mechanism*, and *Rapid View Synchronization*.

2.6. Related Work

There is abundant literature on consensus and leader-based consensus in specific (e.g., [15, 17, 21, 25, 37, 53, 66, 102, 113, 114, 140, 141, 146, 151, 167]), to reduce the communication cost and improve performance and resilience of the consensus systems [27, 42, 101, 103, 108, 117, 142, 145, 148, 165]. In previous sections, we already discussed how SPOTLESS relates to PBFT [39, 40], RCC [77], HOTSTUFF [164], and NARWHAL-HS [49]. Next, we shall focus on other works that deal with either *improving throughput and scalability* or with *simplifying consensus*, the two strengths of SPOTLESS.

Trusted Hardware There is a large body of work on improving and simplifying leader-based consensus by employing *trusted hardware*. A representative example is MINBFT [160], which uses trusted hardware to prevent malicious leaders from proposing conflicting client requests in a view. Via the usage of trusted hardware, one can reduce the impact of Byzantine behavior and simplify both the normal case and the recovery process, e.g., MINBFT eliminates one round of communication from PBFT and can operate in deployments in which $n > 2f$ (instead of the typical $n > 3f$) holds. Protocols that require trusted hardware place significant restrictions on the environment in which they can run, due to which their practical usage is limited. As such, these protocols are not a replacement for a general-purpose consensus such as SPOTLESS. [83]

Leader-Less Consensus. Leader-less protocols such as HONEYBADGER [123] and DUMBO [68] eliminate the limitations of PBFT and other leader-based consensus protocols via a fully decentralized and fully asynchronous design. In these protocols, all replicas have the same responsibilities, due to which the cost of consensus is equally spread-out over all replicas. These leader-less protocols claim to improve resilience over PBFT in asynchronous environments. Due to the high complexity of fully asynchronous consensus, their practical performance is limited, however.

Sharding. Recently, there have been several approaches toward scalability of RDMSs by sharding them, e.g., [13, 50, 56, 86]. Although sharding has the potential to drastically improve scalability for certain workloads, it does so at a high cost for complex workloads. Furthermore, sharding impacts resilience, as sharded systems put requirements on the number of failures in each individual shard (instead of putting those requirements on all replicas in the system). Finally, sharding is orthogonal to consensus, as proposed sharded systems all require a high-performance consensus protocol to run individual shards. For this task, SPOTLESS is an excellent candidate.

Reducing Primary Costs. There are several approaches toward reducing the cost for the leader to coordinate consensus in PBFT-style leader-based consensus protocols, thereby reducing the limitations of leader-based designs. Examples include (1) protocols such as FASTBFT [112] and the geo-scale aware GEOBFT [80] that use a *hierarchical communication architecture* to offload the cost for a leader to propose client requests to other replicas; (2) protocols such as NARWHAL-HS [49] that use a gossip-based communication protocol to replicate client requests, thereby sharply reducing the cost for leaders to propose these requests; and (3) protocols such as ALGORAND [61] that restrict consensus to a small subset of the replicas in the system (whom then enforce their decisions upon all other replicas), thereby reducing the cost of the leader to coordinate consensus. Unfortunately, each of these protocols introduces its own added complexity or environmental restrictions to the design of consensus, e.g., FASTBFT requires trusted hardware, the design of GEOBFT impacts resilience in a similar way as sharded designs do, and ALGORAND relies on complex cryptographic primitives due to which it can only guarantee to work with high probability.

2.7. Conclusion

In this chapter, we proposed SPOTLESS, a high-performance robust consensus protocol. SPOTLESS combines the high throughput of *concurrent consensus architectures* with the reduced complexity provided by *streamlined rotational consensus*. Furthermore, SPOTLESS improves on the resilience of existing streamlined rotational consensus designs by introducing the *Rapid View Synchronization protocol*, which guarantees a continuous low-cost recovery path that is robust during unreliable communication and does not require costly threshold signatures.

We have put the design of SPOTLESS to the test by implementing it in APACHE RESILIENTDB, our high-performance resilient fabric, and we compared SPOTLESS with existing consensus protocols. Our experiment results show that the performance of SPOTLESS is excellent: SPOTLESS greatly outperforms traditional leader-based consensus protocols such as PBFT by up to 430%, NARWHAL-HS by up to 137%, and HOTSTUFF by up to 3803%. SPOTLESS is even able to outperform RCC, a state-of-the-art high-throughput concurrent consensus protocol, by up to 23% in optimal conditions, while providing lower latency in all cases. Furthermore, SPOTLESS can maintain a stable latency and consistently high throughput, even during failures.

HotStuff-1: Linear Consensus with One-Phase Speculation

Chapter 2 focused on boosting throughput and recovery cost in rotational consensus. Latency, however, remains a first-order concern for blockchain applications in which clients wait for transaction finality. In this chapter, we turn to *client-perceived latency* and ask: *how early can replicas safely respond to clients without violating safety?* The material in this chapter appeared as [93].

Stable-leader BFT protocols of the PBFT family [40, 76] attain low no-failure-case latency through *speculative execution* [66, 74, 102], but they pay for this with quadratic-cost view changes that drop throughput to zero whenever a leader fails. Streamlined linear protocols in the HOTSTUFF family [32, 118, 164] eliminate the heavyweight view change by rotating leaders every view, but they incur extra latency relative to the speculative stable-leader protocols, and the regular leader rotation exposes them to two new attacks: *leader-slowness* (rational leaders delay their proposals to maximize MEV [48, 132, 139]) and *tail-forking* [65] (faulty leaders prevent correct leaders' proposals from committing). Stable-leader and streamlined designs thus appear to face a conundrum: each closes one set of problems while opening another.

HOTSTUFF-1 resolves this conundrum by simultaneously importing speculative execution into a streamlined linear protocol *and* mitigating the attacks that streamlining introduces. HOTSTUFF-1 contributes:

- (1) A novel algorithmic core that combines regular leader rotation with linear communication, streamlining, and *speculative execution*. HOTSTUFF-1 acts as an optimist, executing client requests speculatively and serving clients with the results of uncommitted transactions.
- (2) An *adaptive slotting* algorithm that gives each leader access to multiple slots within its view, mitigating leader-slowness and tail-forking attacks.

Early finality confirmation through speculation. The notion of applying speculative execution to BFT protocols is not new: Castro's PhD thesis [38] introduced *tentative execution* for PBFT, later extended by PoE [74], with further variants in Zyzzyva [102] and SBFT [66]. All of these are

stable-leader protocols and **stop** speculative execution during view changes, since recovery requires re-running consensus on past sequence numbers from the new leader’s state. Streamlined protocols, by contrast, do **not have** the option of stopping consensus and rerunning past transactions, which makes naively importing speculation unsafe. We identify this conundrum as the *Prefix Speculation Dilemma*, and HOTSTUFF-1 is the first streamlined protocol to resolve it by precisely characterizing when a replica may speculatively execute a proposal. Building on HOTSTUFF-2 [118], HOTSTUFF-1 allows replicas to send commit-votes directly to clients as soon as a transaction is prepared and highly likely to commit; on receiving a quorum of $\mathbf{n}-\mathbf{f}$ matching responses, the client learns both the commit decision and the execution result simultaneously, enabling an early finality confirmation.

Low latency through adaptive slotting. Speculation alone resolves the latency gap but not leader-slowness or tail-forking. HOTSTUFF-1 therefore incorporates a novel *slotting* mechanism that gives each leader access to multiple slots, so that a rational leader is incentivized to commit its earlier slots quickly to earn the chance to propose more. Fixing the number of slots per view is insufficient, where a fast leader will simply slow down its last slot. Thus, we introduce an *adaptive* variant in which a leader proposes as many slots as fit within its view. Adaptive slotting in turn raises a new question: how does the next leader know it has received the previous leader’s last slot? We answer this with the notion of *trusted/distrusted* previous leaders, which lets a correct leader propose its first slot at network speed when the previous leader is correct.

Resilience to tail-forking attacks. HOTSTUFF-1 with slotting guarantees that in each view v , if L_v proposes at least two slots, at most one of them can remain uncertified, and it can only be tail-forked if fewer than $\mathbf{f}+1$ correct replicas voted for it. This is achieved via *carry* blocks and a dual-certificate mechanism (*New-View* and *New-Slot* certificates) that enforces the previous slot’s inclusion in the next leader’s first-slot proposal.

We illustrate the practicality of HOTSTUFF-1 (with and without slotting) by implementing it in APACHE RESILIENTDB (incubating) [16] and evaluating against HOTSTUFF and HOTSTUFF-2. HOTSTUFF-1 yields up to 41.5% and 24.2% lower client latency than HOTSTUFF and HOTSTUFF-2, respectively, in the no-failure case, and HOTSTUFF-1 with slotting maintains throughput under leader-slowness and tail-forking attacks where the baselines collapse.

In summary, this chapter makes the following contributions:

- (1) We introduce HOTSTUFF-1, the first speculative, streamlined, linear BFT consensus protocol that serves clients with early finality confirmations.
- (2) We expose the *Prefix Speculation Dilemma* for streamlined BFT protocols and present a tailored resolution.
- (3) We introduce adaptive slotting in HOTSTUFF-1 to mitigate leader-slowness and tail-forking, while guaranteeing no extra delay for subsequent correct leaders.

3.1. Background and System Model

Modern databases require replication to guarantee availability to their clients; consensus protocols help keep these replicas consistent [105, 131]. As consensus can quickly bottleneck system performance, a large body of existing work attempts to optimize these protocols [159]. A majority of existing databases [47, 88, 156] employ *crash fault-tolerant* consensus protocols to guarantee consistent replication despite crash failures [105, 131]. This crash-failure threat model is sufficient for these databases as they are managed by a single organization. In this chapter, 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 [12, 16, 155].

We assume the system model adopted by existing partially synchronous BFT consensus protocols [40, 66, 74, 102, 164]. We assume a system of $\mathbf{n}$ replicas, of which at most $\mathbf{f}$ are faulty (malicious or crash-failed), and the remaining $\mathbf{n} - \mathbf{f}$ replicas are correct; $\mathbf{n} \geq 3\mathbf{f} + 1$. 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 $[1, \mathbf{n}]$ using function $\text{id}(R)$.

Authenticated communication: each client/replica uses digital signatures to sign a message [95]. Additionally, replicas make use of the BLS threshold signature scheme [28] to form $(\mathbf{n}, t)$ 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 $\mathbf{n}$ to create the threshold signature. A receiver can use the corresponding public key to verify whether at least t replicas contributed to this signature. We use the notation $\langle R \rangle_m$ 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 $H(x) = H(x')$ [95].

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 [55] 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 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 .

3.2. 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 §3.5. To illustrate the challenges in introducing speculation to streamlined consensus protocols, we first briefly recap the skeleton of the HOTSTUFF-2 [118] 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 3.1(i)). In each view, a leader

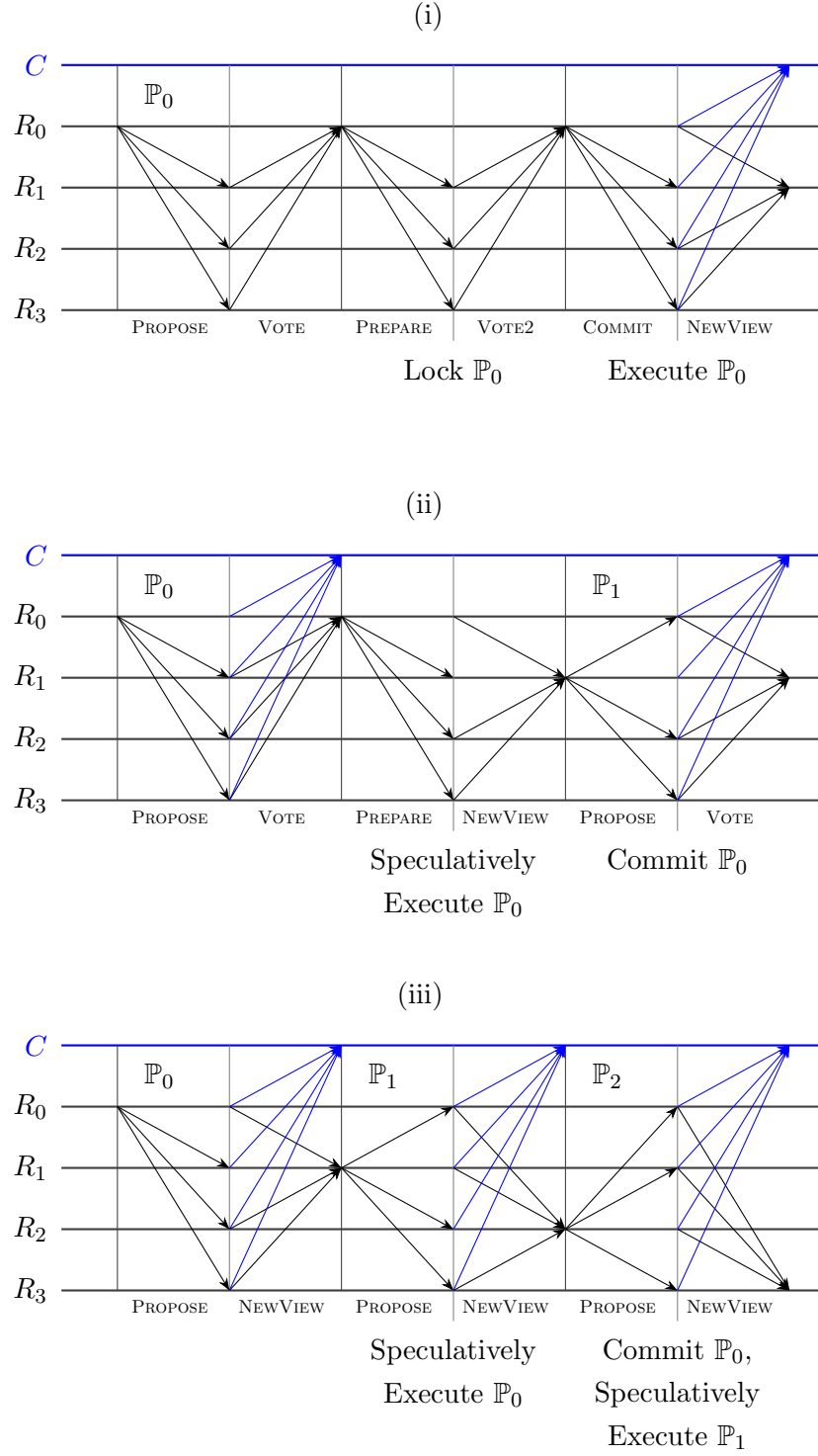


FIGURE 3.1. Workflows of (i) BASIC HOTSTUFF-2, (ii) BASIC HOTSTUFF-1, and (iii) STREAMLINED HOTSTUFF-1

proposes a transaction T and forms consecutive certificates on the initial proposal over two-and-half phases. In the first half-phase, the leader proposes the transaction T . In each subsequent phase:

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

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

Sending early finality confirmations.

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

$\mathbf{n} - \mathbf{f}$ 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 $\mathbf{f} + 1$ 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 $\mathbf{n} - \mathbf{f}$ responses because $\mathbf{f} + 1$ speculative responses only guarantee that one correct replica prepares the transaction. Commitment is guaranteed only when at least $\mathbf{f} + 1$ correct replicas prepare the transaction. Thus, a client learns that a transaction will finalize only upon collecting $\mathbf{n} - \mathbf{f}$ responses. Figure 3.1 (ii) depicts our HOTSTUFF-1 protocol.

Although clients of HOTSTUFF-1 wait for $\mathbf{f}$ 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 §3.6, 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 speculation¹: 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

¹See Section 3.8.1 for a detailed explanation of why the dilemma breaks *safety*.

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 .

Note on Speculation in Stable-Leader Protocols.

As stated in the introduction, the notion of applying speculative execution to BFT protocols is not new [38, 66, 74, 102]. 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 3.2.1. ***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 3.2.2. **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 3.3.4), then R must perform a *rollback* operation in the local-ledger. R can observe that at least $\mathbf{n} - \mathbf{f}$ 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 §3.3.2.

3.3. Speculative Core

We first describe the variant of basic (non-streamlined) HOTSTUFF-1 variant; in §3.4, we describe the streamlined HOTSTUFF-1.

3.3.1. 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 $\mathbf{n} - \mathbf{f}$ replicas for its transaction T , it records this set of responses as an early finality confirmation for T , marks T as *executed* and accepts the result of execution.

Replica pseudocode. In Figure 3.2, we present the pseudo-code for basic HOTSTUFF-1. Prior to describing the algorithm in detail, we lay down some useful definitions.

DEFINITION 3.3.1. 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 $\mathcal{C}(v)$ for a proposal m aggregates $\mathbf{n} - \mathbf{f}$ threshold signature-shares for $\mathcal{P}(v)$ in view v .*

Local state (replica R) :

- 1: $\mathcal{P}(v_{lp})$, v_{lp} : highest known prepare-cert and view; $\mathcal{C}(v_{lc})$, v_{lc} : highest known commit-cert and view
- 2: v : current view; $\mathcal{T}$: pending uncommitted blocks; local-ledger and global-ledger

Leader role (running at leader L_v) :

- 3: **event** Upon PACEMAKER.ENTerview(v) **do**
- 4: Wait until received **n – f** NEWVIEW messages for view v
- 5: Wait until received $\mathcal{P}(v - 1)$ **or** **n** NEWVIEW messages **or** PACEMAKER.SHARETIMER(v)
- 6: Let B_v be a yet-to-be-proposed block of client transactions
- 7: Broadcast $m = \langle L_v \rangle_{\text{PROPOSE}(B_v, v, \mathcal{P}(v_{lp}), \mathcal{C}(v_{lc}))}$
- 8: **event** Received **n – f** NEWVIEW messages with shares for $\mathcal{P}(v - 1)$ **do**
- 9: $\mathcal{C}(v - 1) \leftarrow \text{CREATETHRESHOLDSIGN}(\mathbf{n} - \mathbf{f} \text{ shares})$
- 10: **event** Received **n – f** PROPOSEVOTE messages **do**
- 11: $\mathcal{P}(v) \leftarrow \text{CREATETHRESHOLDSIGN}(\mathbf{n} - \mathbf{f} \text{ shares})$
- 12: Broadcast $\langle L_v \rangle_{\text{PREPARE}(v, \mathcal{P}(v))}$

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

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

FIGURE 3.2. Basic HOTSTUFF-1.

DEFINITION 3.3.2. 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 3.3.3. 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 3.3.4. 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, $\mathcal{C}(v_{lc})$, 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 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 $\mathcal{C}(v - 1)$ (Line 9) and updates $\mathcal{C}(v_{lc})$. After learning the highest certificate across all correct replicas (Line 5, see further explanations in §3.3.2), the leader aggregates client transactions (yet to be proposed) into a block B_v ² and creates a PROPOSE message m that includes the view number v , B_v , $\mathcal{P}(v_{lp})$, and $\mathcal{C}(v_{lc})$. Then, L_v broadcasts m to all replicas (Lines 6-7). *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 L_v (Line 13), 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. (§3.3.2).

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 L_v (Lines 13-17). Otherwise, R ignores the message.

²Since block-level execution is atomic—meaning that either all transactions within a block are executed or none are—we use the terms “block B ” and “transaction T ” interchangeably throughout the remainder of this chapter.

Prepare. When 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 10-11). Then, L_v creates a PREPARE message including $\mathcal{P}(v)$ and broadcasts it (Line 12).

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 18-24):

- (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.

Note on execution model. Once the transactions are ordered, they are executed sequentially. This chapter focuses on reducing client latency caused by consensus. Thus, we assume the simplest execution model: sequential execution of the ordered transactions. Alternatively, other execution designs, such as parallel transaction execution, can be employed, but these require detecting and resolving conflicts among transactions.

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 a NEWVIEW message, which includes $\mathcal{P}(v_{lp})$, and forwards it to the leader L_{v+1} of view $v + 1$. It then invokes the pacemaker to orchestrate view-synchronization as needed (Line 30).

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 3.3.5. Traditional Commit Rule. *A replica marks a block B_{v-1} as committed when it receives a commit-certificate $\mathcal{C}(v - 1)$ for B_{v-1} .*

DEFINITION 3.3.6. Prefix Commit Rule. *A replica marks a block B_{v-1} as committed when it receives a prepare-certificate $\mathcal{P}(v)$ that extends $\mathcal{P}(v - 1)$.*

³We defined No Gap rule for streamlined protocols in § 3.2. 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 .

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 25-26). Next, each replica calls the EXITVIEW procedure. On receiving $\mathbf{n} - \mathbf{f}$ threshold shares for the same prepare-certificate, the leader of the next view combines them into a commit-certificate (Line 9) and forwards it to all the replicas. Upon receiving a commit-certificate $\mathcal{C}(v)$, a replica R adds the block B_v to the global-ledger and marks it committed (Line 14). *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 § 3.3.2.

3.3.2. 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 (§3.3.2.1). 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 27-30).

Lack of certificates from the last view

Leader 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, 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 (§3.3.2.1), after GST , it is guaranteed that by $PACEMAKER.SHARETIMER(v)$, which is 3Δ after L_v enters view v , L_v will receive $NEWVIEW$ messages including known certificates from all correct replicas. Thus, if L_v did not receive $\mathcal{P}(v-1)$, it should wait until either it received $\mathbf{n}$ $NEWVIEW$ messages or $PACEMAKER.SHARETIMER(v)$ (Line 5).

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 3.3.7. 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 22-23).*

See Section 3.8.2 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 $\mathbf{n} - \mathbf{f}$ replicas. If such is the case, neither the client will receive an early finality confirmation for m , nor the replicas will receive a commit-certificate for m in view $v+1$. So, how can we decide the fate of m ?

If m conflicts with another proposal m' speculated in a view $w, w > v$, then it will be rolled back as described in § 3.2. 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 $\mathcal{C}(x)$ for m' (Line 14).

```

1: function SHARETIMER( $v$ ) do
2:   return  $StartTime[v] + 3\Delta$ 

3: function COMPLETEDVIEW() do
4:   if  $v \bmod f + 1 \neq 0$  then
5:     Call ENTERVIEW( $v$ )
6:   else
7:     Call SYNCHRONIZEEPOCH( $v$ )

8: function SYNCHRONIZEEPOCH( $v$ ) do
9:    $\delta_R \leftarrow \text{CREATETHRESHOLD SHARE}(v)$ 
10:  Send  $\langle \text{WISH}(v, \delta_R) \rangle_R$  to leaders  $L_{v+k}, k = 0, 1, 2, \dots, f$ .

Epoch Leader role (running at leader  $L_{v+k}, k = 0, 1, 2, \dots, f$ ) :
11: event Upon receiving  $\mathbf{n} - \mathbf{f}$  WISH messages of view  $v$  do
12:    $TC_v \leftarrow \text{CREATETHRESHOLD SIGNATURE}(\mathbf{n} - \mathbf{f} \text{ distinct } \delta_r \text{ shares})$ 
13:   Broadcast  $TC_v$ .

Epoch Backup role (running at each replica  $R$ ) :
14: event Upon receiving  $TC_v$  at time  $t$  do
15:   Relay  $TC_v$  to the leaders  $L_{v+k}, k = 0, 1, 2, \dots, f$ 
16:   for  $k \leftarrow 0, 1, 2, \dots, f$  do
17:      $StartTime[v+k] \leftarrow t + k\tau$ 
18:   Call ENTERVIEW( $v$ )

```

FIGURE 3.3. Pseudocode of Pacemaker Protocol

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 19), 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 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' .

3.3.2.1. Pacemaker. For a system to make *progress*, at least $\mathbf{n} - \mathbf{f}$ correct replicas should be in the same view. Otherwise, a leader cannot collect enough votes to make progress and to generate a prepare-certificate (§3.4). 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 [42, 108]; group views into *epochs*, each of which contains $\mathbf{f} + 1$ consecutive views, and conduct view synchronization at the beginning of every epoch.

In Figure 3.3, we illustrate the pseudocode for pacemaker. Every time a replica R reaches at the end of a view, it calls the function `COMPLETEDVIEW` (Lines 3-7) to check if the next view (say v) is part of the current epoch. If this is the case, R enters view v . Otherwise, v is the first view of the next epoch ($v \bmod (\mathbf{f} + 1) = 0$) and R must synchronize its view with the other replicas. R calls the function `SYNCHRONIZEVIEW( $v$ )` (Lines 8-10) and delays entering the view v until the view synchronization is complete.

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

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 $t + 3\Delta$.

3.4. Streamlined Speculation

Basic HOTSTUFF-1 (§3.3.1) 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 $\mathbf{n} - \mathbf{f}$ NEWVIEW messages from the preceding view. The leader first attempts to create a prepare-certificate from the threshold shares it received from the replicas. It then selects the highest 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 3.3.6) 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 §3.2. 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, enforcement of the No-Gap rule (Definition 3.2.2) is necessary, that is, $w = v - 1$. See Section 3.8.3 for examples of not following the Prefix Speculation rule and the No Gap in streamlined HOTSTUFF-1.

3.4.1. 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 3.1 (iii).

Propose. When the leader L_v for view v receives well-formed NEWVIEW messages from at least $\mathbf{n} - \mathbf{f}$ replicas (Figure 3.4 Line 2), it tries to combine their threshold signature-shares into a

Leader role (running at leader L_v) :

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

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

- 8: **event** Received $\langle L_v \rangle_{\text{PROPOSE}(B_v, \mathcal{P}(w))}$ **do**
- 9: **if** $\mathcal{P}(w)$ extends $\mathcal{P}(w - 1)$ *$\triangleright$ commit-rule* **then**
- 10: Execute all transactions up to (incl.) B_{w-1} , add result to global-ledger and respond to clients
- 11: **if** $w = v - 1$ *$\triangleright$ No-Gap rule* **then**
- 12: **if** predecessor of $\mathcal{P}(v - 1)$ is in global-ledger *$\triangleright$ 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 *$\triangleright$ 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 $\langle R \rangle_{\text{NEWVIEW}(v+1, \mathcal{P}(w), \delta_R)}$ to L_{v+1}
- 19: Call EXITVIEW()
- 20: **event** Upon timeout **do**
- 21: Send $\langle R \rangle_{\text{NEWVIEW}(v+1, \mathcal{P}(v_{lp}), \perp)}$ to L_{v+1} .
- 22: Call EXITVIEW()
- 23: **function** EXITVIEW() **do**
- 24: $v \leftarrow v + 1$. *$\triangleright$ disable voting for view v*
- 25: Call PACEMAKER.COMPLETEDVIEW()

FIGURE 3.4. Streamlined HOTSTUFF-1.

threshold signature to create a certificate $\mathcal{P}(v - 1)$ for view $v - 1$ (Line 7). If L_v fails, it keeps waiting for more NEWVIEW messages until it forms a certificate $\mathcal{P}(v - 1)$ or it receives $\mathbf{n}$ 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 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, 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 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 Section 3.7 for the correctness proof.

3.5. 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 [48, 132, 139]. 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.

EXAMPLE 3.5.1. *Assuming that each block can include at most 100 transactions and the maximum allowed time for a view to complete is 4s, while it takes a leader approximately 1s to create a block and ensure that its proposal completes all phases of HOTSTUFF-1. In an ideal case, the latency for each transaction would be $\approx 1s$. A rational leader will wait for four seconds to create the block in the hope of selecting the top 100 highest fees paying transactions, which ensures the average latency to be $\approx 4s$.*

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 decision.

EXAMPLE 3.5.2. *Assuming that R_0 and R_1 are the leaders for views v and $v + 1$, respectively, and R_1 is malicious. In view v , R_0 broadcasts a PROPOSE message for B_v containing $\mathcal{P}(v - 1)$, and all replicas send a PROPOSEVOTE message for B_v to R_1 . As R_1 is malicious, in view $v + 1$, assume that R_1 initiates the tail-forking attack by ignoring the NEWVIEW messages for B_v and broadcasts a PROPOSE message for B_{v+1} that includes the certificate $\mathcal{P}(v - 1)$. Since no replica has access to a higher known certificate than $\mathcal{P}(v - 1)$, all replicas accept B_{v+1} , create a threshold signature-share for B_{v+1} , and send it with a NEWVIEW message to R_2 . Consequently, all the work done during view v is a waste.*

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, provided that at least $f + 1$ correct replicas have voted for it.

With slotting, assuming Example 3.5.1, we expect each leader to propose at least four blocks (one per slot) per view with latency $\approx 1s$; assuming Example 3.5.2, R_1 can only tail-fork the last slot of view v , but three out of four blocks will reach consensus.

3.5.1. 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 3.5, 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 fv , indicating the view in which it was formed, i.e., it is formed by L_{fv} .

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 3.6) 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 .

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 $\mathbf{n} - \mathbf{f}$ NEWVIEW messages sent to L_v , e.g., in Figure 3.5, $B_{1,2}$ extends $\mathcal{P}(4, 1)$; or (ii) **extend** its highest certificate $\mathcal{P}(s_{lp}, v_{lp})$ and **carry** a block B_u .

DEFINITION 3.5.3. Carry Block: The lowest uncertified block B_u that extends the certificate $\mathcal{P}(s_{lp}, v_{lp})$.

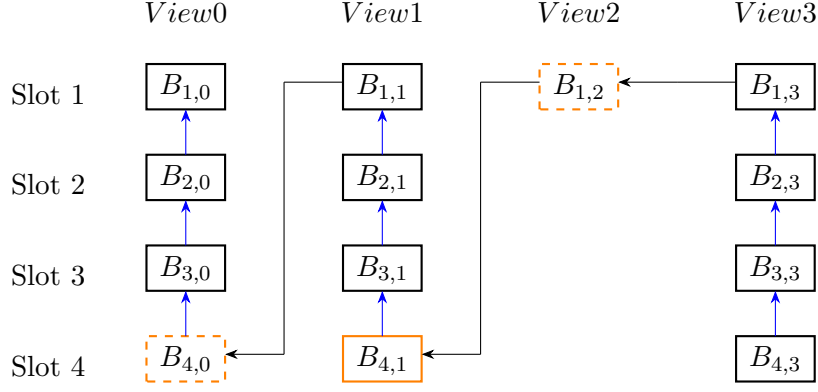


FIGURE 3.5. Chain in HotStuff-1 with Slotting, in which solid black blocks are extended with *New-Slot* certificates. To provide a *self-contained proof* of “no tail-forking” in first-slot proposals, solid orange blocks are *extended* in way (i), with a *New-View* certificate formed by the next leader; while shaded orange blocks are *carried* in way (ii), without a certificate.

- 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 L_v proposed at least two slots in view v and $\mathbf{f} + 1$ correct replicas have voted for its last slot, then view- v slots are protected from *tail-forking attacks*. See further explanations in § 3.5.2.

Figure 3.6 and 3.7 illustrate the pseudocode of streamlined HOTSTUFF-1 with slotting.

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

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

Thus, 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 $\mathbf{n} - \mathbf{f}$ replicas. 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) L_v received **n** NEWVIEW messages.
- (3) $\text{PACEMAKER.SHARETIME}(v)$.
- (4) For any slot higher than $\mathcal{P}(s_{lp}, v_{lp})$, 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 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 § 3.5.3.

If condition (1) is satisfied, 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, L_v broadcasts a PROPOSE message m containing the block $B_{1,v}$, a batch of new transactions, and the updated certificate $\mathcal{P}(s_{lp}, v_{lp})$. For example, in Figure 3.5, L_1 proposes $B_{1,1}$ extending the *New-View* certificate $\mathcal{P}(4, 0)$ (see Figure 3.6, Lines 9–10).

If (2)-(4) is satisfied, L_v proposes its first-slot proposal in way (ii): 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 3.5, 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 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 $\mathcal{P}(s-1, v)$, 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 3.7, 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).

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 $B_{s_w+1,w}$.

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 L_v) :

- 4: **event** Upon PACEMAKER.ENTREVIEW() **do**
- 5: Keep updating $\mathcal{P}(s_{lp}, v_{lp})$ while receiving $\mathbf{n} - \mathbf{f}$ NEWVIEW messages
- 6: Wait until (1) formed a *NEW-VIEW* certificate $\mathcal{P}(s_w, w)$ **or** (2) received $\mathbf{n}$ NEWVIEW messages **or** (3) PACEMAKER.SHARETIMER(v) **or** (4) received $\mathbf{n} - k, 1 \leq k \leq \mathbf{f}$, NEWVIEW messages, but there are fewer than $\mathbf{f} + 1 - k$ votes for any slot higher than (s_{lp}, v_{lp})
- 7: **if** 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 $m = \langle L_v \rangle_{\text{PROPOSE}(B_{1,v}, 1, v, \mathcal{P}(s_w, w), \perp)}$
- 11: **else**
- 12: $B_u \leftarrow$ the lowest uncertified block that extends $\mathcal{P}(s_{lp}, v_{lp})$
- 13: Broadcast $m = \langle L_v \rangle_{\text{PROPOSE}(B_{1,v}, 1, v, \mathcal{P}(s_{lp}, v_{lp}), H_u)}$
- 14: **event** Received $\mathbf{n} - \mathbf{f}$ NEWVIEW messages with *NEW-VIEW* signature-shares of $\mathcal{P}(s_w, w), w < v$ **do**
- 15: $\mathcal{P}(s_w, w) \leftarrow \text{CREATENEWVIEWTHRESHOLDSIGN}(\mathbf{n} - \mathbf{f} \text{ distinct } \delta_h \text{ shares}, f v = v)$
- 16: **event** Received $\mathbf{n} - \mathbf{f}$ NEWSLOT messages with *NEW-SLOT* signature-share of $\mathcal{P}(s, v)$ **do**
- 17: $\mathcal{P}(s, v) \leftarrow \text{CREATENEWSLOTTHRESHOLDSIGN}(\mathbf{n} - \mathbf{f} \text{ distinct } \delta_R \text{ shares})$
- 18: Let $B_{s+1,v}$ be a block of client transaction yet to be proposed
- 19: Broadcast $m = \langle L_v \rangle_{\text{PROPOSE}(B_{s+1,v}, s+1, v, \mathcal{P}(s, v), \perp)}$
- 20: **event** Received from a *trusted* leader 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 L_{v-1} a NEWVIEW message with a lower certificate that is formed in view $v - 1$ **then**
- 24: Mark L_{v-1} as *distrusted*.

FIGURE 3.6. Additional Local State and Leader Role in Streamlined HotStuff-1 with Slotting.

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,

```

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  $\mathcal{P}(s_w, w).fv = v$   $\triangleright$ 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
    $B_u.view = \mathcal{P}(s_w, w).fv$   $\triangleright$ 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  $B_u.view = w$   $\triangleright$ 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  $w = v$   $\triangleright$ Case 4 then
10:    return true
11:  return false

  Backup role (running at each replica  $R$  (including leader)) :
12: event Received  $\langle L_v \rangle_{\text{PROPOSE}(B_{s,v}, \mathcal{P}(s_w, w), H_u)}$  do
13:   if  $\mathcal{P}(s_w, w)$  extends  $\mathcal{P}(s_w - 1, w)$   $\triangleright$ 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  $\mathcal{P}(s_w-1, w-1)$   $\triangleright$ 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  $v = w + 1$ )  $\triangleright$ No-Gap rule
   and predecessor of  $\mathcal{P}(s_w, w)$  is in global-ledger  $\triangleright$ 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  $\langle R \rangle_{\text{NEWSLOT}(s, v, \mathcal{P}(s_{lp}, v_{lp}), \delta_R)}$  to  $L_v$ 
24:   else
25:     Send  $\langle R \rangle_{\text{REJECT}(s, v, \mathcal{P}(s_{lp}, v_{lp}))}$  to  $L_v$ 
26:    $s \leftarrow s + 1$   $\triangleright$ 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  $\langle R \rangle_{\text{NEWVIEW}(v+1, \mathcal{P}(s_{lp}, v_{lp}), H_h, \delta_h)}$  to  $L_{v+1}$ 
30:    $v \leftarrow v + 1, s \leftarrow 1$   $\triangleright$ disable voting for view v
31:   Call PACEMAKER.COMPLETEDVIEW()

```

FIGURE 3.7. Backup Role in Streamlined HOTSTUFF-1 + Slotting.

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 3.3.6) 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) $s = 1, v = w + 1$.

3.5.2. 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 L_v , where $B_{s, v}$ extends $B_{s-1, v}$. It is guaranteed that if L_v proposed at least two slots and at least $\mathbf{f} + 1$ 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 $\mathbf{f}+1$ correct replicas have voted for $B_{s, v}$, it becomes impossible to form a *New-View* certificate $\mathcal{P}(s-1, v)$, as those $\mathbf{f}+1$ 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*.

3.5.3. Advancing at Network Speed with Trusted Previous Leaders. Generally, leaders of BFT consensus must guarantee they extend a highest certificate that all correct 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: 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 L_{v-1} fails to broadcast its final slot to at least $\mathbf{n}-\mathbf{f}$ well-behaving replicas before their view timers expire, the next leader 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 L_{v-1} that includes a certificate formed in view $v-1$ (Figure 3.6, Line 20), i.e., a *New-Slot* certificate of view $v-1$ or a *New-View* certificate with $fv = v-1$, 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 L_{v-1} when exiting view $v-1$, assuming a certificate was formed in view $v-1$. The trusted/distrusted mechanism thus enables L_v to propose its first-slot block at network speed, avoiding unnecessary delays.

However, a Byzantine previous leader L_{v-1} , initially *trusted* by L_v , may conceal the highest certificate it has formed and sent to other correct replicas. This can cause 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 L_v containing its highest certificate (Figure 3.7, Line 25). If L_v had previously received a NEWVIEW message from L_{v-1} containing a lower certificate formed in view $v-1$ (Figure 3.6, Line 23), it then marks L_{v-1} as *distrusted*. In future views where L_v becomes leader again, it no longer trusts L_{v-1} and follows the four conditions described in §3.5.1 when entering the view. As a result, each malicious leader 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 L_v can learn the highest certificate known to any correct replica. If condition (1) holds, then at least $\mathbf{f}+1$ 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 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.

If some correct replica holds a certificate $\mathcal{P}(s^*, v^*)$ higher than the leader's $\mathcal{P}(s_{lp}, v_{lp})$, then at least $\mathbf{f} + 1$ correct replicas have voted for B_{s^*, v^*} and will vote for a block not lower than B_{s^*, v^*} in the NEWVIEW messages sent to the leader. While processing the NEWVIEW messages, if condition (1) or (4) is satisfied, then no block higher than $\mathcal{P}(s_{lp}, v_{lp})$ can get more than $\mathbf{f}$ correct-replica votes through the NEWVIEW messages, thus such $\mathbf{f} + 1$ correct replicas do not exist; If condition (2) or (3) is satisfied, then the higher certificate will be received by the leader, as the pacemaker protocol guarantees that after *GST*, all NEWVIEW messages from correct replicas can arrive at the leader by $\text{SHARETIMER}(v)$.

3.6. 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 $\mathbf{f}$ 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) [16]; C++20 code with Google Protobuf v3.10.0 for serialization and NNG v1.5.2 for networking. APACHE RESILIENTDB is an optimized blockchain framework that provides APIs to implement a new consensus protocol. As threshold signature algorithms are expensive and can quickly bottleneck the computational resources, the leader sends a list of $\mathbf{n} - \mathbf{f}$ digital signatures (from distinct replicas) as a certificate.

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

(1) **HOTSTUFF.** 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 [54] and TPC-C [1]:

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

(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 ($\mathbf{f}+1$ for HOTSTUFF/HOTSTUFF-2 and $\mathbf{n}-\mathbf{f}$ for HOTSTUFF-1) for that transaction.

3.6.1. Scalability. Impact of the number of replicas In Figures 3.8 (a) and (b), we present various system metrics as a function of the number of replicas; we increase the number of replicas from $\mathbf{n} = 4$ to $\mathbf{n} = 64$.

As expected, an increase in the number of replicas causes a proportional decrease in the throughput for all the protocols due to an $O(\mathbf{n})$ 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.

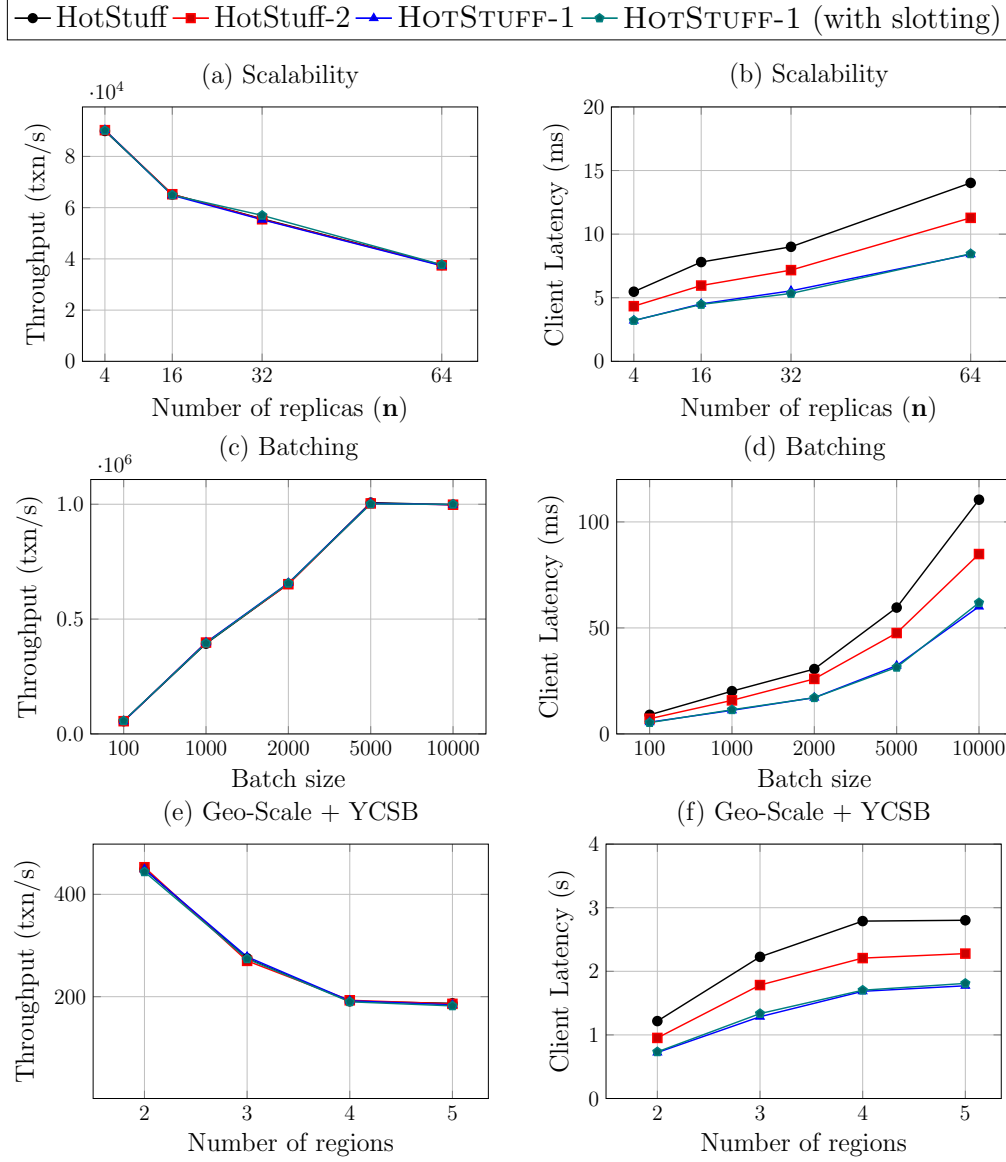


FIGURE 3.8. Scalability: impact of number of replicas, batch size, and geo-distribution (YCSB).

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,

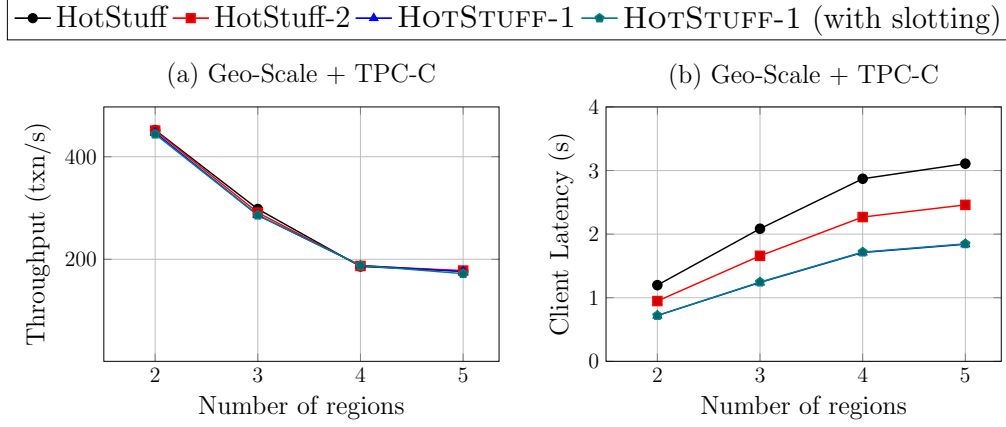


FIGURE 3.9. Geo-distributed scalability (TPC-C benchmark).

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 3.8 (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 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 3.8 (e-f) and 3.9, 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.

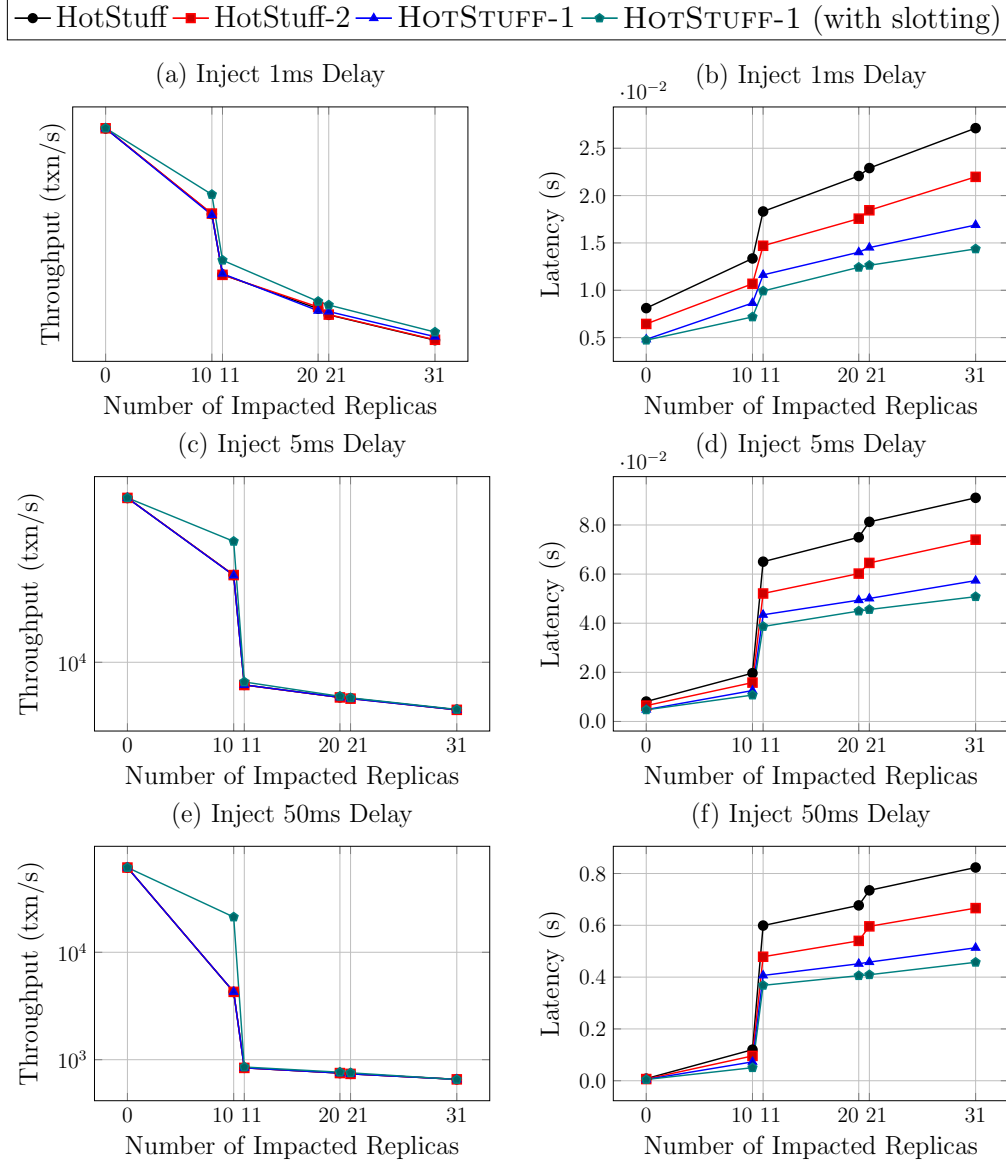


FIGURE 3.10. Performance with injected message delays (1ms, 5ms, 50ms).

3.6.2. 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)

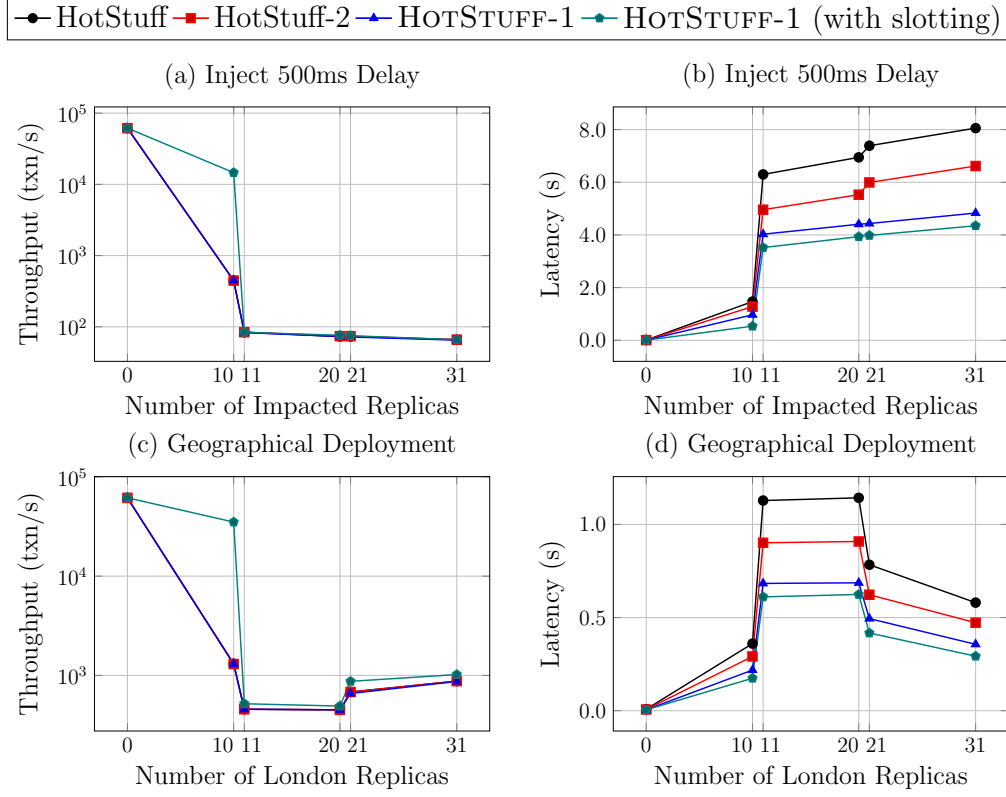


FIGURE 3.11. Performance with 500ms injected delay and geographical deployment.

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 3.10 (a–f) and 3.11 (a–b) 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 $k \geq \mathbf{n}-\mathbf{f}$, clients can receive at most $\mathbf{f}$ responses from non-impacted replicas, causing latency to be dominated by the slower, impacted replicas.

When $k \leq \mathbf{f}$, 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 $\mathbf{n} = 31$ 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 $k \in \{0, \mathbf{f}, \mathbf{f}+1, \mathbf{n}-\mathbf{f}-1, \mathbf{n}-\mathbf{f}, \mathbf{n}\}$. The results, shown in Figures 3.11(c) and (d), illustrate the impact of increasing geographic separation among replicas.

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

3.6.3. Failure Resiliency. Leader slowness phenomenon.

We now study the impact of leader slowness (§3.5) on streamlined protocols by varying the number of slow leaders from 0 to $\mathbf{f}$, with $\mathbf{n} = 32$ 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 3.12(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 $\mathbf{f} = 10$ 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%

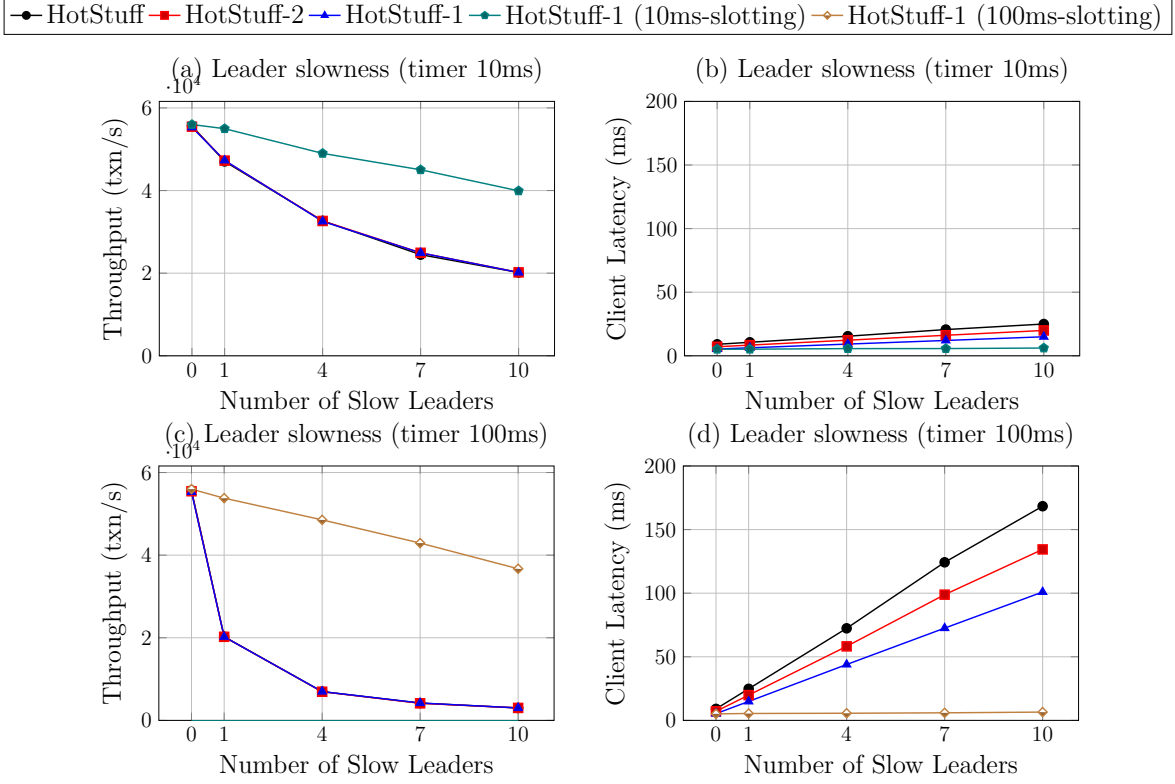


FIGURE 3.12. Impact of leader slowness on throughput and latency.

higher latency with 1 and $\mathbf{f} = 10$ 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 (§3.5). In this experiment, we vary the number of faulty leaders from 0 to $\mathbf{f}$, using $\mathbf{n} = 32$ replicas and a batch size of 100. As shown in Figures 3.13(a) and (b), 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 $\mathbf{f} = 10$ 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

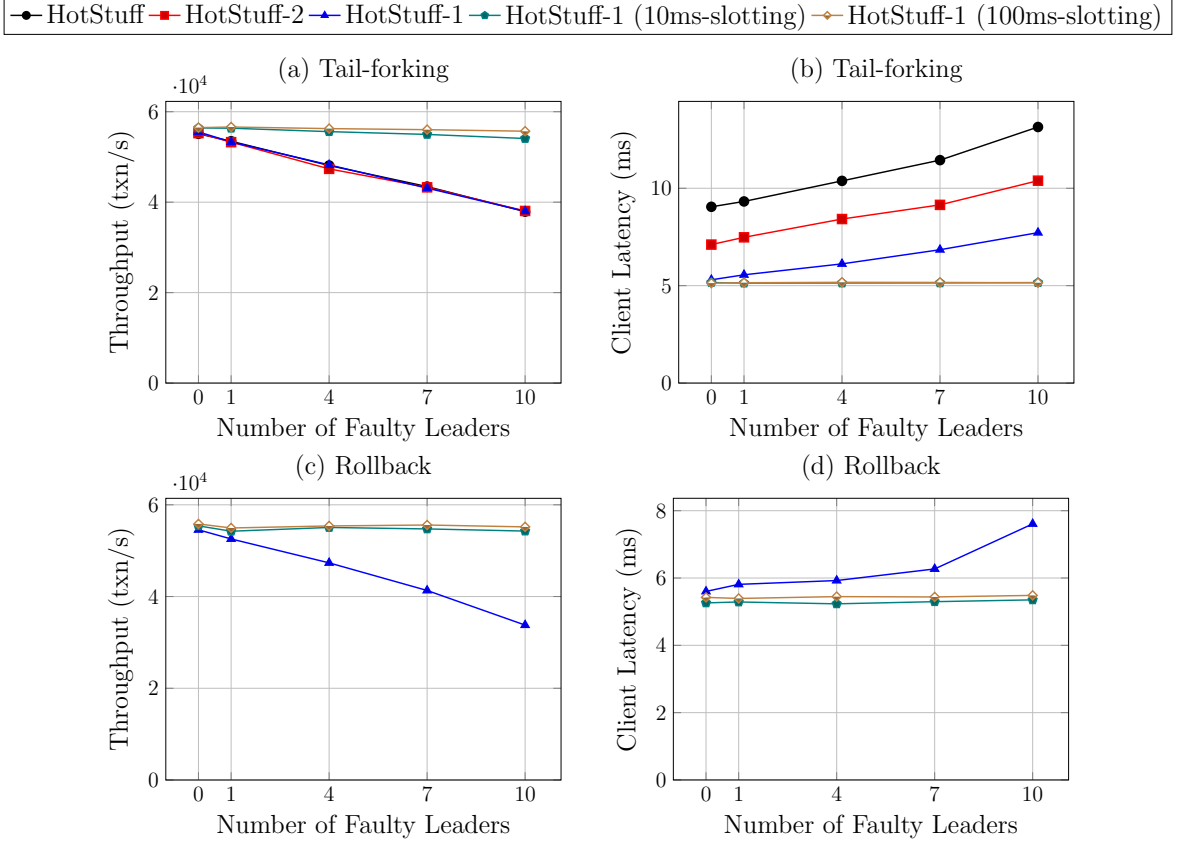


FIGURE 3.13. Impact of tail-forking and rollback attacks.

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 $\mathbf{f} = 10$ faulty leaders.

Rollback. In HOTSTUFF-1, speculation on uncommitted transactions may require replicas to roll back speculated transactions. In Figures 3.13(c)–(d), we vary the number of faulty leaders from 0 to $\mathbf{f}$ and allow each to force up to $\mathbf{f}$ correct replicas to roll back transactions, using $\mathbf{n} = 32$ replicas and a batch size of 100. Notably, in HOTSTUFF-1 with *slotting*, a faulty leader 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 $\mathbf{f} = 10$, 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.

3.7. Correctness Proofs

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

LEMMA 3.7.0.1. *Let R_1 and R_2 be two correct replicas that execute blocks B_v^1 and B_v^2 of view v . If $\mathbf{n} = 3\mathbf{f} + 1$, then $B_v^1 = B_v^2$.*

PROOF. A correct replica R_i executes a block B_v^i only after obtaining a prepare certificate for B_v^i , as specified in Figure 3.4, which consists of threshold signature shares from $\mathbf{n} - \mathbf{f}$ replicas.

Let S_i denote the set of replicas that voted for the proposal containing B_v^i , so $|S_i| = \mathbf{n} - \mathbf{f} = 2\mathbf{f} + 1$. Let $X_i = S_i \setminus \mathbf{f}$ represent the subset of correct replicas in S_i . Since at most $\mathbf{f}$ replicas may be faulty, we have $|X_i| \geq 2\mathbf{f} + 1 - \mathbf{f} = \mathbf{f} + 1$.

Assume, for the sake of contradiction, that $B_v^1 \neq B_v^2$. This implies that $X_1 \cap X_2 = \emptyset$, since any common correct replica would not vote for two distinct blocks in the same view. Therefore, the combined set $X_1 \cup X_2$ would contain at least $2(\mathbf{f} + 1) = 2\mathbf{f} + 2$ correct replicas.

However, this contradicts the total number of correct replicas in the system, which is $\mathbf{n} - \mathbf{f} = 2\mathbf{f} + 1$. Thus, our assumption must be false, and we conclude that $B_v^1 = B_v^2$. $\square$

LEMMA 3.7.0.2. *If a replica R receives a certificate $\mathcal{P}(v+1)$ that extends certificate $\mathcal{P}(v)$, then for any view $w > v$, no certificate $\mathcal{P}(w)$ that conflicts with $\mathcal{P}(v)$ can exist.*

PROOF. We know that a replica R received $\mathcal{P}(v+1)$ that extends $\mathcal{P}(v)$, which is only possible if $\mathbf{n} - \mathbf{f} = 2\mathbf{f} + 1$ replicas that set $\mathcal{P}(v)$ as their higher known certificate also voted for $\mathcal{P}(v+1)$. Let's denote the $\mathbf{f} + 1$ correct replicas from these $\mathbf{n} - \mathbf{f}$ replicas as A . Further, certificate $\mathcal{P}(w)$ conflicts with $\mathcal{P}(v)$, $w > v$, which implies that $\mathcal{P}(v)$ and $\mathcal{P}(w)$ extend the same ancestor and $\mathcal{P}(w)$ received support of $\mathbf{n} - \mathbf{f} = 2\mathbf{f} + 1$ replicas. Let's denote the $\mathbf{f} + 1$ correct replicas from these $\mathbf{n} - \mathbf{f}$ replicas as A' . As $w \neq v + 1$, so $w > v + 1$. Moreover, any correct replica that sets $\mathcal{P}(v)$ as its highest known certificate will not vote for a conflicting block. Thus, $|A| + |A'| = 2\mathbf{f} + 2$, which is more than the total number of correct replicas and is a contradiction. $\square$

COROLLARY 3.7.1. *If $\mathbf{f} + 1$ correct replicas speculatively execute a block B_v , then no higher-view block B_w with $w > v$ that conflicts with B_v can be committed.*

PROOF. By Lemma 3.7.0.2, if a correct replica speculatively executes a block B_v , it must have observed a valid certificate $\mathcal{P}(v)$ for B_v and set it as its highest known certificate.

If $\mathbf{f} + 1$ correct replicas have speculatively executed B_v , then at least $\mathbf{f} + 1$ correct replicas have locked on $\mathcal{P}(v)$. Since there are only $2\mathbf{f} + 1$ correct replicas in total, no quorum of $\mathbf{n} - \mathbf{f} = 2\mathbf{f} + 1$ votes can be collected for any conflicting block B_w with $w > v$, as at least $\mathbf{f} + 1$ correct replicas will refuse to vote for any certificate conflicting with B_v .

Thus, no conflicting certificate can be formed at a higher view, and therefore no conflicting block can be committed. $\square$

LEMMA 3.7.0.3. *If a correct replica R commits a block B_v , then no conflicting block can be committed.*

PROOF. Assume, for contradiction, that there exists a block B_w proposed in view $w > v$ that conflicts with B_v , and that another correct replica R' has committed B_w . This would imply that the global ledgers at replicas R and R' have diverged, violating safety.

For both B_v and B_w to be committed, replicas R and R' must have followed the prefix commit rule described in §3.4:

- R must have received a certificate $\mathcal{P}(v + 1)$ that extends $\mathcal{P}(v)$, thereby committing B_v .
- R' must have received a certificate $\mathcal{P}(w + 1)$ that extends $\mathcal{P}(w)$, thereby committing B_w .

Since $w > v$ and $w \neq v + 1$, it follows that $w > v + 1$. However, by Lemma 3.7.0.2, once certificates $\mathcal{P}(v)$ and $\mathcal{P}(v + 1)$ are formed, no conflicting certificate $\mathcal{P}(w)$ for $w > v$ can be constructed. Therefore, the assumption that B_w was committed by R' leads to a contradiction.

Hence, once a correct replica commits B_v , no conflicting block can be committed. $\square$

THEOREM 3.7.1. (**Safety**) *Streamlined HOTSTUFF-1 guarantees consensus safety in a system with $\mathbf{n} \geq 3\mathbf{f} + 1$ replicas: if two correct replicas R_1 and R_2 commit blocks B_1 and B_2 , respectively, at the same position k in the global ledger, then $B_1 = B_2$.*

PROOF. By Lemma 3.7.0.3, once a correct replica commits a block, no conflicting block can be committed. Therefore, both B_1 and B_2 are permanently part of the respective global ledgers of R_1 and R_2 , and must not conflict.

Suppose, for the sake of contradiction, that $B_1 \neq B_2$ and that B_2 extends B_1 . Then, B_1 must occupy position k in the global ledger of R_1 , and also appear as part of the prefix of B_2 . This would

imply that the prefix of B_2 contains $k - 1$ blocks (including B_1), while the prefix of B_1 contains at most $k - 2$ blocks. However, this contradicts the assumption that B_1 is committed at position k with $k - 1$ blocks in its prefix.

Consequently, B_1 and B_2 must be of the same view and must be the same block, and we conclude that no two correct replicas can commit different blocks at the same position. Hence, Streamlined HOTSTUFF-1 guarantees consensus safety. $\square$

Next, we prove the liveness guarantee of streamlined HOTSTUFF-1. Following prior works [118, 164], we assume the existence of a Global Stabilization Time (GST) and an appropriately chosen view timer length τ , such that correct replicas eventually overlap in the same view after view synchronization. This assumption ensures that the timer is long enough for the leader to process NEWVIEW messages, obtain the highest known certificate, propose a block, and for replicas to respond with votes. We denote by v_s the first synchronized view after GST.

LEMMA 3.7.1.1. *For the leader L_v of view v , where $v \geq v_s$, its proposal will be supported by all correct replicas.*

PROOF. Assume that L_v enters view v at time t . According to [42, 108], the PaceMaker ensures that all correct replicas enter view v by $t + 2\Delta$. Thus, L_v can receive NEWVIEW messages from all correct replicas by $t + 3\Delta$. If L_v forms a certificate $\mathcal{P}(v - 1)$, then it is guaranteed to be the highest certificate known to any correct replica; otherwise, the highest certificate can still be learned from the received NEWVIEW messages.

The proposal sent by L_v will arrive at all correct replicas by $t + 4\Delta$. By setting a sufficiently long timer, all correct replicas remain in view v upon receiving the proposal and will vote for it. $\square$

LEMMA 3.7.1.2. *Assume three consecutive correct leaders: L_v , L_{v+1} , and L_{v+2} , with $v \geq v_s$. If L_v proposes a block B_v in view v , then all correct replicas will commit B_v in view $v + 2$.*

PROOF. By Lemma 3.7.1.1, in view v , block B_v will be supported by all correct replicas, allowing L_{v+1} to form a certificate $\mathcal{P}(v)$.

Similarly, in view $v + 1$, block B_{v+1} extending $\mathcal{P}(v)$ will be supported by all correct replicas, enabling L_{v+2} to form $\mathcal{P}(v + 1)$.

Then, in view $v + 2$, all correct replicas will receive block B_{v+2} extending $\mathcal{P}(v + 1)$, which satisfies the prefix commit rule for B_v . Hence, B_v will be committed by all correct replicas in view $v + 3$. $\square$

THEOREM 3.7.2. (*Liveness*) *All correct replicas eventually commit a transaction T .*

PROOF. Since there are $\mathbf{n} = 3\mathbf{f} + 1$ replicas and HOTSTUFF-1 rotates leaders in a round-robin manner, there must exist a set of three consecutive correct leaders: L_v , L_{v+1} , and L_{v+2} , with $v \geq v_s$. By Lemma 3.7.1.2, any transaction T contained in the block proposed in view v will eventually be committed by all correct replicas. $\square$

COROLLARY 3.7.2. *Assume two consecutive correct leaders: L_v and L_{v+1} , with $v \geq v_s$. If L_v proposes a block B_v in view v , then B_v will eventually be committed.*

PROOF. Consider the setting in Lemma 3.7.1.2, stopping at two consecutive correct leaders: L_v and L_{v+1} . In view $v + 2$, all correct replicas will eventually receive a block B_{v+1} that extends $\mathcal{P}(v)$ and will adopt $\mathcal{P}(v)$ as their highest known certificate. This ensures that in any future view, no conflicting certificate with $\mathcal{P}(v)$ can be formed.

By Theorem 3.7.2, we know that there will eventually be a set of three consecutive correct leaders—say L_w , L_{w+1} , and L_{w+2} , with $w \geq v + 1$ —who commit a block B_w that extends the chain containing B_v , because B_v is a lower-view non-conflicting block of B_w . Since B_v is in the prefix of that chain, all correct replicas will eventually commit B_v . $\square$

COROLLARY 3.7.3. (*Client Safety*) *If a client receives $\mathbf{n} - \mathbf{f}$ matching responses for transactions in a block B_v , then B_v will eventually be committed by all correct replicas.*

PROOF. By Lemmas 3.7.0.2 and 3.7.0.3, we derive this result as follows:

If a client receives $\mathbf{n} - \mathbf{f}$ responses for transactions in block B_v , then at least $\mathbf{f} + 1$ of these responses must have come from correct replicas. There are two cases to consider:

- (1) **Speculative execution:** At least $\mathbf{n} - \mathbf{f}$ replicas speculatively executed B_v and replied to the client. This quorum must include at least $\mathbf{f} + 1$ correct replicas. By Corollary 3.7.1, this implies that no higher-view certificate conflicting with B_v can be formed, and thus no conflicting block can be committed.

- (2) **Committed execution:** At least one correct replica committed and executed B_v . By Lemma 3.7.0.3, no conflicting block can be committed.

In either case, once the client receives $\mathbf{n} - \mathbf{f}$ matching responses for B_v , it is guaranteed that no conflicting block can later be committed.

By Theorem 3.7.2, we know that there will eventually be a set of three consecutive correct leaders—say L_w , L_{w+1} , and L_{w+2} , with $w \geq v + 1$ —who commit a block B_w that extends the chain containing B_v , because B_v is a lower-view non-conflicting block of B_w . Since B_v is in the prefix of that chain, all correct replicas will eventually commit B_v . $\square$

3.8. Speculation Safety and Rollback Examples

3.8.1. Speculation Safety in Basic-HotStuff-1. Allowing replicas to speculatively execute transactions in a proposal m upon receiving a certificate for m is insufficient to guarantee safety for clients, i.e., a client mistakenly considers a transaction as committed after receiving $\mathbf{n} - \mathbf{f}$ responses for it. The following examples demonstrate how speculative execution after observing a prepare certificate may violate safety unless both the **Prefix Speculation Rule** and the **No Gap Rule** are strictly followed.

Prefix Speculation Rule: We first present a scenario where the Prefix Speculation Rule is violated. Assume the highest certificate across all replicas is $\mathcal{P}(0)$, and the total number of replicas is $\mathbf{n} = 3\mathbf{f} + 1$. Partition the $2\mathbf{f} + 1$ correct replicas into three disjoint sets: A , A' , and A^* , such that $|A| = |A'| = \mathbf{f}$ and $|A^*| = 1$. Suppose the first four leaders are Byzantine.

- In view 1, the leader L_1 proposes block B_1 extending $\mathcal{P}(0)$. A quorum of $\mathbf{n} - \mathbf{f}$ replicas supports this proposal by sending threshold signature shares for B_1 , allowing L_1 to form the prepare certificate $\mathcal{P}(1)$. However, L_1 forwards $\mathcal{P}(1)$ only to the $\mathbf{f}$ replicas in set A , who speculatively execute B_1 and respond to the client.

- In view 2, leader L_2 disregards $\mathcal{P}(1)$ and instead proposes a new block B_2 extending $\mathcal{P}(0)$ to all replicas. Replicas in A' and A^* support B_2 , allowing L_2 to form $\mathcal{P}(2)$, which is forwarded only to A' . The A' replicas then speculatively execute B_2 and respond to clients.

- In view 3, L_3 ignores $\mathcal{P}(2)$ and proposes B_3 extending $\mathcal{P}(1)$ to all replicas. Replicas in A and A^* support B_3 , enabling the formation of $\mathcal{P}(3)$, which is sent only to A^* . Upon receiving it,

A^* replicas speculatively execute both B_3 and its ancestor B_1 (not following the Prefix Speculation Rule).

- In view 4, L_4 disregards $\mathcal{P}(3)$ and proposes B_4 extending $\mathcal{P}(2)$ to all replicas. Replicas in A and A' support B_4 , leading to the formation of $\mathcal{P}(4)$. Although $\mathcal{P}(2)$ conflicts with the highest known certificate $\mathcal{P}(1)$ known to replicas in A , they are required to support B_4 due to the higher view number of $\mathcal{P}(2)$. L_4 then broadcasts $\mathcal{P}(4)$ to all replicas.

Ultimately, B_4 becomes the highest known certificate across all replicas and will eventually be committed.

- However, an unsafe scenario for clients arises: the client for transactions in B_1 may have received $\mathbf{n} - \mathbf{f}$ responses from replicas in A , A^* , and $\mathbf{f}$ faulty replicas, even though B_1 will not be committed.

This example illustrates the Prefix Speculation dilemma: replicas vote to commit a block B_v along with its prefix, but cannot safely speculate on the prefix unless specific conditions are met. According to the Prefix Speculation rule (Definition 3.2.1), speculative execution is safe only when the prefix of B_v is already committed. In this example, replicas in A^* vote to commit a block B_3 along with its prefix B_1 , but also speculate on the prefix B_1 , violating the Prefix Speculation rule. Thus, the client forms a commit-vote quorum for B_1 consisting of A , A^* , and $\mathbf{f}$ faulty replicas. However, the replicas in A , which voted for B_1 in view 1, will switch to support a higher conflicting block B_2 after receiving $\mathcal{P}(2)$ in view 4, which makes the quorum invalid.

From prior literature on speculative consensus protocols, we note that Zyzzyva adopts a useful approach: replicas attach a view number to their speculative results, and clients are required not to aggregate responses from different views. This practice can help mitigate the risk of inconsistency caused by speculative execution across views.

No Gap Rule: Secondly, we present a scenario where the No Gap Rule is violated, with the same assumption as we had in the previous scenario.

- In view 1, the leader L_1 proposes block B_1 extending $\mathcal{P}(0)$. A quorum of $\mathbf{n} - \mathbf{f}$ replicas supports this proposal by sending threshold signature shares for B_1 , allowing L_1 to form the prepare certificate $\mathcal{P}(1)$. However, L_1 forwards $\mathcal{P}(1)$ only to the $\mathbf{f}$ replicas in set A , who speculatively execute B_1 and respond to the client.

- In view 2, leader L_2 disregards $\mathcal{P}(1)$ and instead proposes a new block B_2 extending $\mathcal{P}(0)$ to all replicas. Replicas in A' and A^* support B_2 , allowing L_2 to form $\mathcal{P}(2)$, which is forwarded only to A' . The A' replicas then speculatively execute B_2 and respond to clients.

- In view 3, L_3 ignores $\mathcal{P}(2)$ and proposes B_3 extending $\mathcal{P}(1)$ to A^* only. Upon receiving B_3 extending $\mathcal{P}(1)$, A^* replicas speculatively execute B_1 (not following the No Gap Rule). L_3 does not collect votes and then no certificate is formed.

- In view 4, L_4 proposes B_4 extending $\mathcal{P}(2)$ to all replicas. All replicas support B_4 because it extends the highest certificate $\mathcal{P}(2)$. L_4 then broadcasts $\mathcal{P}(4)$ to all replicas. Ultimately, B_4 becomes the highest known certificate across all replicas and will eventually be committed.

- However, an unsafe scenario for client arises: the client for transactions in B_1 may have received $\mathbf{n} - \mathbf{f}$ responses from replicas in A , A^* , and $\mathbf{f}$ faulty replica, even though B_1 will not be committed.

This example highlights the critical requirement that when a replica R wishes to speculate on a block B_v , it must ensure there is **no view gap** between the view in which the prepare-certificate was formed and its current view. This is necessary to prevent speculative execution on a proposal that may be superseded by a higher certificate formed during the gap—one that remains unknown to R . In this example, for replica A^* in view 3, a view gap exists between its current view and the view in which L_1 formed its certificate; meanwhile, a higher certificate $\mathcal{P}(2)$, formed by L_2 in the gap, will later supersede it.

According to the No Gap Rule (Definition 3.2.2), in BASIC HOTSTUFF-1, if R wishes to speculatively execute a block B_w , it is safe only if $w = v$ and $\mathcal{P}(w)$ is formed in view v .

In summary, the general intuition behind both the Prefix Speculation Rule and the No-Gap Rule is the same: *a replica should not speculate on a block if there may exist a higher conflicting certificate that could supersede it*. The Prefix Speculation Rule emphasizes this principle for blocks in the uncommitted prefix, while the No-Gap Rule focuses on the block of the latest received certificate.

With the two rules in place, receiving $\mathbf{n} - \mathbf{f}$ responses implies that at least $\mathbf{f} + 1$ correct replicas have speculatively executed the block. This, in turn, implies that $\mathbf{f} + 1$ correct replicas are locked on the certificate of the speculated block, which ensures that no higher conflicting certificate can be formed, thereby guaranteeing safe speculation.

3.8.2. Rollback is Necessary. Providing early finality confirmation responses is inherently speculative. If a conflicting certificate is later formed at a higher view, replicas must roll back their local-ledger state to maintain safety. We illustrate this necessity with the following scenario.

Assume the system starts in the initial state $\perp$. In view 1, the leader L_1 proposes a message B_1 that extends $\mathcal{P}(\perp)$. A quorum of $\mathbf{n} - \mathbf{f}$ replicas supports the proposal by sending threshold signature shares, allowing L_1 to form the prepare certificate $\mathcal{P}(1)$. This certificate is forwarded to a subset of $\mathbf{f}$ correct replicas, denoted by set A . The replicas in A speculatively execute the transactions in B_1 and respond to the client.

Now suppose the leader of view 2, L_2 , is also faulty and ignores the highest known certificate $\mathcal{P}(1)$. It proposes a conflicting message B_2 extending $\mathcal{P}(\perp)$ and broadcasts it to all replicas. A distinct set of $\mathbf{n} - \mathbf{f}$ replicas, disjoint from A , support B_2 , allowing L_2 to form the conflicting certificate $\mathcal{P}(2)$. L_2 then broadcasts $\mathcal{P}(2)$ to all replicas.

Upon receiving $\mathcal{P}(2)$, the replicas in set A detect that it was formed at a higher view than $\mathcal{P}(1)$, and consequently roll back their local ledger state. After the rollback, all correct replicas speculatively execute transactions in B_2 and respond to the client. Once the client receives responses from $\mathbf{n} - \mathbf{f}$ replicas, the transactions in B_2 are considered committed by the clients.

3.8.3. Speculation Safety in Streamlined-HotStuff-1. The following examples demonstrate, in Streamlined HOTSTUFF-1, how speculative execution after observing a prepare certificate may violate safety unless both the **Prefix Speculation Rule** and the **No Gap Rule** are strictly followed.

Prefix Speculation Rule: We first present a scenario where the Prefix Speculation Rule is violated. Assume the initial highest certificate across all replicas is $\mathcal{P}(0)$, and the total number of replicas is $\mathbf{n} = 3\mathbf{f} + 1$. Partition the $2\mathbf{f} + 1$ correct replicas into three disjoint sets: A , A' , and A^* , such that $|A| = |A'| = \mathbf{f}$ and $|A^*| = 1$. Suppose the first eight leaders are Byzantine.

- In view 1, the leader L_1 proposes block B_1 that extends $\mathcal{P}(0)$. A set of $\mathbf{n} - \mathbf{f}$ replicas support this proposal by sending their threshold signature shares for B_1 to the leader of view 2, L_2 , enabling it to form the prepare certificate $\mathcal{P}(1)$. Assume that L_2 proposes block B_2 extending $\mathcal{P}(1)$, but only forwards this certificate to a subset A of $\mathbf{f}$ correct replicas. These replicas in A speculatively execute B_1 and reply to the client.

- In view 3, L_3 proposes block B_3 extending $\mathcal{P}(0)$ and sends it to replicas in sets A' and A^* . These replicas support B_3 , enabling the leader of view 4, L_4 , to form a certificate $\mathcal{P}(3)$. Assume L_4 then proposes block B_4 extending $\mathcal{P}(3)$ and forwards it to set A' . The replicas in A' speculatively execute B_3 and respond to the client.

- In view 5, the leader L_5 ignores the higher certificate $\mathcal{P}(3)$ and proposes B_5 , which extends the lower certificate $\mathcal{P}(1)$, to all replicas. Sets A and A^* support B_5 , allowing L_6 (view 6) to form a new certificate $\mathcal{P}(5)$. L_6 forwards $\mathcal{P}(5)$ only to set A^* , whose replicas speculatively execute B_5 and its prefix B_1 (not following the Prefix Speculation Rule).

- In view 7, L_7 disregards the highest known certificate $\mathcal{P}(5)$ and proposes B_7 extending $\mathcal{P}(3)$ to all replicas. Sets A and A' support B_7 , enabling L_8 to form certificate $\mathcal{P}(7)$. Although $\mathcal{P}(3)$ conflicts with the highest known certificate $\mathcal{P}(1)$ known to replicas in A , they must support B_7 due to the higher view number of $\mathcal{P}(3)$. L_8 broadcasts $\mathcal{P}(7)$ to all replicas.

Eventually, $\mathcal{P}(7)$ becomes the highest known certificate of all correct replicas and will be committed.

- However, an unsafe scenario for clients arises: the client for transactions in B_1 may have received $\mathbf{n} - \mathbf{f}$ responses from replicas in A , A^* , and $\mathbf{f}$ faulty replicas, even though B_1 will never be committed.

This example illustrates the Prefix Speculation dilemma: replicas vote to commit a block B_v along with its prefix, but cannot safely speculate on the prefix unless specific conditions are met. According to the Prefix Speculation rule (Definition 3.2.1), speculative execution is safe only when the prefix of B_v is already committed. In this example, replicas in A' vote to commit a block B_5 along with its prefix B_1 , but also speculate on the prefix B_1 , violating the Prefix Speculation rule. Thus, the client forms a commit-vote quorum for B_1 consisting of A , A^* , and $\mathbf{f}$ faulty replicas. However, the replicas in A , which voted for B_1 in view 2, will switch to support a higher conflicting block B_3 after receiving $\mathcal{P}(3)$ in view 7, which makes the quorum invalid.

No Gap Rule: Secondly, we present a scenario where the No Gap Rule is violated, with the same assumption as we had in the previous scenario.

- In view 1, the leader L_1 proposes block B_1 that extends $\mathcal{P}(0)$. A set of $\mathbf{n} - \mathbf{f}$ replicas support this proposal by sending their threshold signature shares for B_1 to the leader of view 2, L_2 , enabling it to form the prepare certificate $\mathcal{P}(1)$. Assume that L_2 proposes block B_2 extending $\mathcal{P}(1)$, but

only forwards this certificate to a subset A of $\mathbf{f}$ correct replicas. These replicas in A speculatively execute B_1 and reply to the client.

- In view 3, L_3 proposes block B_3 extending $\mathcal{P}(0)$ and sends it to replicas in sets A' and A^* . These replicas support B_3 , enabling the leader of view 4, L_4 , to form a certificate $\mathcal{P}(3)$. Assume L_4 then proposes block B_4 extending $\mathcal{P}(3)$ and forwards it to set A' . The replicas in A' speculatively execute B_3 and respond to the client.

- In view 5, L_5 proposes B_5 , which extends $\mathcal{P}(1)$, to only set A^* , whose replicas speculatively execute B_1 (not following the No Gap Rule).

- In view 6, L_6 proposes B_6 extending $\mathcal{P}(3)$ to all replicas. All replicas support B_7 , enabling L_7 to form certificate $\mathcal{P}(6)$. Although $\mathcal{P}(3)$ conflicts with the highest known certificate $\mathcal{P}(1)$ known to replicas in A and A^* , they must support B_6 due to the higher view number of $\mathcal{P}(3)$. L_7 broadcasts $\mathcal{P}(6)$ to all replicas.

Eventually, $\mathcal{P}(7)$ becomes the highest known certificate across all correct replicas and will be committed.

- However, an unsafe scenario for clients arises: the client for transactions in B_1 may have received $\mathbf{n} - \mathbf{f}$ responses from replicas in A , A^* , and $\mathbf{f}$ faulty replicas, even though B_1 will never be committed.

This example highlights the critical requirement that when a replica R wishes to speculate on a block B_v , it must ensure there is **no view gap** between the view in which the prepare-certificate was formed and its current view. This is necessary to prevent speculative execution on a proposal that may be superseded by a higher certificate formed during the gap—one that remains unknown to R . In this example, for replica A^* in view 5, a view gap exists between its current view and the view in which L_1 formed its certificate; meanwhile, a higher certificate $\mathcal{P}(3)$, formed by L_4 in the gap, will later supersede it.

According to the No Gap Rule (Definition 3.2.2), in Streamlined HOTSTUFF-1, if R wishes to speculatively execute a block B_w , it is safe only if $w = v - 1$ and $\mathcal{P}(w)$ is formed in view v .

3.9. Related Work

Extensive literature exists on consensus, with numerous studies (e.g., [13, 14, 15, 17, 21, 22, 25, 37, 53, 66, 81, 102, 113, 114, 123, 130, 140, 141, 146, 167]) focused on enhancing consensus systems [26, 27, 70, 75, 80, 83, 84, 87, 101, 103, 116, 117, 121, 136, 145, 148, 165, 168].

Speculation. Protocols belonging to the PBFT family [5, 66, 102] 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 [74, 85]. 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 [118] 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 [89] and Jolteon [60] have quadratic complexity in view-change; AAR [8] employs expensive zero-knowledge proofs; Wendy [62] relies on a new aggregate signature construction (and is super-linear); Marlin [154] 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 [69, 77], MirBFT [151], and SpotLess [94]. These protocols focus mostly on increasing *throughput*, and a majority of them have a HOTSTUFF-core. Thus, their designs are orthogonal to this chapter. Any reduction in latency, the elimination of leader slowness phenomena, and tail-forking attacks will improve them. Autobahn [63] 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 [41, 49, 97, 98, 119, 149, 150, 162] decouple data dissemination from consensus by leveraging *reliable broadcast (RBC)* mechanisms [30]. 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 [18, 23, 147] 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 [115, 127, 128, 161] have $O(n^3)$ worst-case message complexity. RareSync [42] and Lewis-Pye [108] reduce the worst-case message complexity to $O(n^3)$ but face $O(n\Delta)$ latency in the presence of faulty leaders. Fever [109] removes the $O(n\Delta)$ latency but assumes a synchronous start of replicas. Lumiere [110] eliminates the need for the assumption and maintains all other properties of Fever. SpotLess [94] adopts a rapid view synchronization mechanism similar to FastSync [161], 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 [48, 132, 139]. Prior work has illustrated that in Ethereum, for 59% of blocks, proposers have earned higher MEV rewards than block rewards [132], and any additional delay in proposing can help maximize their MEVs [144]. 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 [20]. (ii) Assign block rewards proportional to the number of attestations; a delayed block will receive fewer attestations and thus reduced block rewards [143]. However, if MEV rewards exceed total block rewards, the proposer makes a profit despite losing any block reward.

Tail-forking attack. As described earlier, BeeGees [65] describes the problem of tail-forking. They present an elegant solution to this problem by requiring an incoming leader to collect $\mathbf{n} - \mathbf{f}$ 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 [24], Flow Networks [58], Meter [122] have expressed a latency-over-everything emphasis.

Early adopters of HOTSTUFF, DiemBFT [51], and Aptos that uses a two-phase variant of DiemBFT, Ditto [60], 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.

3.10. Conclusion

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

CHAPTER 4

Carry the Tail in Consensus Protocols

Chapter 3 introduced HOTSTUFF-1, which uses adaptive slotting and a dual-certificate mechanism to mitigate leader-slowness and to bound tail-forking damage to at most one slot per view. Slotting, however, requires that each leader proposes more than one block, and the dual-certificate machinery is tightly coupled to HOTSTUFF-1’s slotting mechanism. In this chapter, we ask a more fundamental question: *can we eliminate tail-forking without slotting and as a drop-in mechanism for any HS-like protocol, while preserving HotStuff’s hallmark linear communication complexity?* The material in this chapter appeared as [78, 79].

The conventional fix for tail-forking, exemplified by BeeGees [65] and MonadBFT [90], sacrifices that linearity: an incoming leader is required either to collect a quorum of $2f+1$ votes for the previous tail or to **prove** that no such quorum can form. BeeGees implements the proof either with quadratic communication or with computationally prohibitive SNARKs; MonadBFT incurs quadratic communication after timeout and asks replicas to send duplicate votes to two different leaders. Either way, the linear-communication tenet that distinguishes the HS-like protocols is broken.

Our Contribution: Carry. We introduce **Carry**, a *practical*, drop-in mechanism for any HS-like protocol that preserves linear communication complexity and restores progress under hostile or sluggish leadership. The key insight is to treat replica votes as *first-class citizens*: in HS-like protocols replicas already *guard their lock* during view changes, but they do *not* guard their vote, which is exactly what a Byzantine leader exploits to fork the tail. Carry makes the vote first-class by carrying it forward across views: if the next leader cannot collect $2f+1$ votes for the previous tail, it must *reinstate* that tail (in full) inside its own proposal. A Byzantine leader can therefore no longer skip a safe block silently; a correct future leader, conversely, can rescue a previous safe block even without a fresh $2f+1$ quorum, closing the gap that BeeGees pays for with quadratic proofs.

The Carry-the-Tail Solution. We present a full atomic-broadcast solution called Carry-the-Tail that incorporates Carry into HotStuff-2. Carry-the-Tail proceeds view by view. At the start of each view, each replica sends the incoming leader its votes (or signed empty shares, if it abstained) for each of the most recent ρ views; the total payload across $\mathbf{n}$ replicas is $O(\mathbf{n} \cdot \rho)$. The leader then issues a proposal that extends the highest pending previous proposal, using the collected votes to justify only the views it skips:

- (1) **Easy case:** If $2\mathbf{f}+1$ votes for the highest previous proposal are available, the leader collects them into a new QC and extends it, with no further evidence needed.
- (2) **Hard case:** If no new QC exists, prior protocols fall back to quadratic communication to prove the absence of higher certificates. Carry instead reinstates (in full) the single highest vote that extends the leader’s highest known QC, and aggregates $2\mathbf{f}+1$ empty vote shares into an *empty certificate* for each intervening view. All such attachments remain bounded by $O(\mathbf{n} \cdot \rho)$.

A replica accepts the proposal as justified if, for each view between the extended QC and the current view, the proposal either reinstates one block or carries an empty certificate.

Carry-the-Tail Properties. The key technical guarantee is that only a consecutive run of more than ρ Byzantine leaders can prevent a correct leader’s proposal from being reinstated or extended:

Definition 1 (ρ -tail-resilience). *We say that a proposal T is ρ -isolated if its view is situated among a succession of ρ consecutive bad leaders. After GST, each proposal T by a correct leader that is not ρ -isolated is guaranteed to be included in the global sequence.*

By the pigeonhole principle, a rotation of $3\mathbf{f}+1$ leaders contains $2\mathbf{f}+1 - (\mathbf{f}/\rho)$ leaders that satisfy ρ -tail-resilience even against $\mathbf{f}$ adversarially interleaved bad leaders. For $\rho = 6$, only a $\frac{1}{12}$ fraction of correct-leader proposals can be tail-forked in the worst case, and a run of six consecutive bad leaders is highly unlikely in practice when leader rotation is randomized.

ρ -tail-resilience is weaker than BeeGees’s *Any-Honest-Leader-Commit* (AHL), which requires every correct-leader block to eventually commit. We deliberately accept this weakening in exchange for retaining HS-like linearity. Carry-the-Tail still guarantees the standard liveness property:

Definition 2 (*Liveness*). After GST, an unbounded number of blocks proposed by correct leaders are committed (despite potential tail-forking attacks).

Carry-the-Tail achieves ρ -tail-resilience and liveness while maintaining the HS-like protocols' defining cost bound:

Definition 3 (*Linearity*). *After GST, a commit decision incurs $O(\mathbf{f}_a \cdot \mathbf{n})$ word-communication cost in the face of $\mathbf{f}_a$ actual faults, and every sequence of $O(\mathbf{n})$ commit decisions incurs $O(\mathbf{n}^2)$ total communication.*

Discussion. Beyond Byzantine tail-forking, Carry also helps with benign **stragglers**, leaders that are slow due to environmental issues. Because every new leader either reinstates its highest vote or attaches an empty certificate, a proposal from a slow leader can still be reinstated and eventually committed even if it received fewer than $2\mathbf{f}+1$ votes in its original view. In summary, Carry restores HotStuff's trademark efficiency while closing the leader-performance gap that has driven recent quadratic remedies.

4.1. Background

This chapter focuses on protocols that solve Byzantine Fault Tolerant atomic broadcast (a.k.a. BFT consensus), in the standard partial synchrony model with $\mathbf{n} = 3\mathbf{f}+1$ replicas, where at most $\mathbf{f}$ replicas are Byzantine [93, 118, 164]. We use streamlined HotStuff-2 [118] as the concrete HS-like protocol on which we exemplify Carry; Carry can be applied to other streamlined block-based protocols.

Blocks and Metadata. A HS-like protocol works through views. In each view v , one replica is designated as the leader and is responsible for proposing a block. We denote the leader of view v by L_v and its proposed block by B_v ; conversely, we denote the view v of a block by $\text{view}(B_v)$.

A key ingredient in the protocol is (a leader) forming a *quorum-certified* block:

- A *Quorum Certificate (QC)* is a cryptographic proof that a block is approved by a quorum of replicas. It is formed by aggregating $2\mathbf{f}+1$ valid votes (signature shares¹) from replicas in a system of $\mathbf{n} \geq 3\mathbf{f}+1$.
- QCs ensure safety by proving that a quorum of replicas agreed on a proposal. We will interchangeably denote the QC for a proposal from view v by $\text{QC}(B_v)$ or $\text{QC}(v)$.

¹To generate and aggregate signature shares, HS-like protocols make use of threshold signatures schemes.

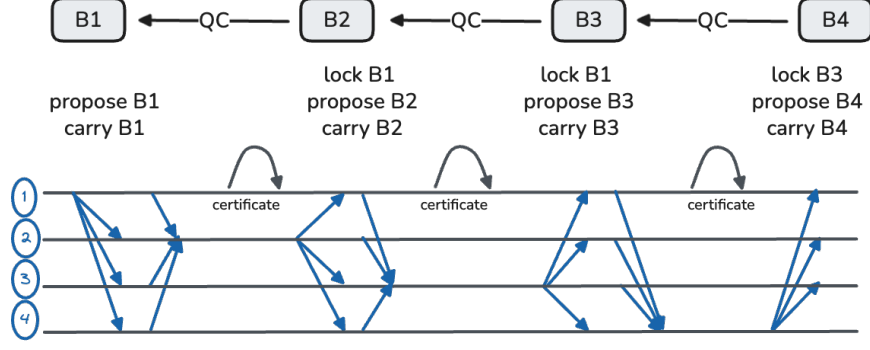


FIGURE 4.1. Streamlined HotStuff-2 protocol flow in a failure-free scenario.

Leaders form a QC in HotStuff-2 with linear communication overhead by borrowing a technique from [34, 137] : a sender that (i) disseminates a block proposal, (ii) collects certified acknowledgments (votes) that are signed with signature-shares, and (iii) aggregates these shares via threshold cryptography. The sender can disseminate the QC to all replicas at a linear word-communication cost.

Blocks are chained to one another through QCs and other metadata. This allows replicas to detect conflicts (e.g., forks) and enforce voting rules to maintain consistency:

- Each block B extends a previously certified block B' by including the certificate $QC(B')$ as $B.qc$.
- Unique to Carry, a block B may *reinstate* an uncertified previous block B' , without having a QC for it, by embedding it (in full) as a *reinstate block* in $B.reinstate$. By slight abuse of terminology, we also say that B extends B' .
- The transitive closure of the two “extend” relationships is denoted $\succeq$.

4.1.1. Streamlined HotStuff-2 in a Nutshell. We use streamlined HotStuff-2 [118], a variant of HotStuff that reduces its latency by two-half phases, to exemplify the Carry mechanism. The workflow of HotStuff-2 is depicted in Figure 4.1. *Note:* Carry can be applied to other streamlined and block-based protocols.

The key safety rules of the protocol are defined below.

Leader Proposal Rule. A leader for view v may propose a block once any of the following conditions are met:

- (1) It forms a fresh quorum certificate $QC(w)$ by aggregating $2f+1$ NEW-VIEW messages with identical votes for some prior view $w < v$.
- (2) It receives $3f+1$ NEW-VIEW messages or the Pacemaker indicates that sufficient time has passed to collect messages from all correct replicas.

These conditions ensure safety by extending the highest known QC, and liveness by allowing the leader to eventually propose even under partial synchrony.

Voting Rule. A replica follows a Voting Rule to decide whether to accept and vote for a leader's proposal. This decision ensures that all correct replicas remain consistent; they only vote for proposals that do not conflict with previously locked blocks.

Specifically, each replica maintains a *lock*, which is the highest QC it has seen. A replica votes for a proposed block B if and only if $B.qc$ has a higher or equal view than its lock. Then it updates its lock to $B.qc$.

Commit Rule. A Commit Rule in HotStuff-2 requires a chain of two consecutive QCs to commit a proposal. The replica commits a block B_v to the total order if it receives B_{v+1} and B_{v+2} , with $B_{v+2}.qc = B_{v+1}$ and $B_{v+1}.qc = B_v$.

Protocol Flow. Algorithm 1 provides a pseudo-code description of HotStuff-2. The general flow of the protocol is view by view, as depicted in Figure 4.1, is as follows:

Replica $\rightarrow$ incoming leader: At the beginning of a new view, each replica sends the incoming leader a NEW-VIEW message that includes:

- (1) the next view number,
- (2) the replica's highest QC (its *lock*), and
- (3) its vote as per the Voting Rule.

Leader $\rightarrow$ replicas: The leader sends replicas a proposal according to the Leader Proposal Rule, that consists of a block B_v that extends the highest QC it aggregated.

Voting: Next, replicas decide whether to accept the proposal based on the Voting Rule. At the end of each view, the replicas proceed to perform the handover protocol with the next incoming leader.

Algorithm 1 (Streamlined) HotStuff-2 Protocol

```
1: State: highest-vote and lock (the QC attached to highest-vote).
2:
3: Begin view  $v$ :
4: if replica is  $L_v \triangleright \text{leader}$  then
5:   wait until  $\text{LEADERPROPOSALCONDITIONS}(v)$ ; create  $B_v$ ;  $\text{ATTACHMETADATA}(B_v)$ ; propose  $B_v$ 
6:
7: On receiving proposal  $B_v$ :
8: if  $\text{VOTINGRULE}(B_v)$  then
9:    $\text{lock} \leftarrow \text{view}(B_v.qc)$ ;  $\text{highest-vote} \leftarrow B_v$ 
10:  $\text{Pacemaker.advanceView}()$ ;  $\text{COMMITRULE}(B_v)$ 
11:
12: On  $\text{Pacemaker.exitView}()$ : advance to view  $v$ ; send NEW-VIEW to next leader with (i) current view, (ii) lock, (iii) signature-share on highest-vote.
13:
14: VotingRule( $B_v$ ):  $\text{view}(\text{lock}) \leq \text{view}(B_v.qc)$ .
15:
16: CommitRule( $B_v$ ): if  $B_v.qc = \text{QC}(B_{v-1})$  and  $B_{v-1}.qc = \text{QC}(B_{v-2})$ , commit the prefix up to (incl.)  $B_{v-2}$ .
17:
18: LeaderProposalConditions( $v$ ): (i) a fresh QC formed from  $2f+1$  NEW-VIEWS with identical highest votes for some  $w < v$ ; (ii)  $3f+1$  NEW-VIEWS received or  $\text{Pacemaker.syncView}()$ .
19:
20: AttachMetadata( $B_v$ ):  $B_v.qc \leftarrow$  highest known QC.
```

The pseudo-code in Algorithm 1 makes use of a **Pacemaker** API: a replica invokes $\text{Pacemaker.advanceView}()$ when it wishes to exit the current view; $\text{Pacemaker.exitView}()$ notifies a replica that the current view has expired; $\text{Pacemaker.syncView}()$ notifies the replica that under synchrony conditions, all other replicas have entered the next view and their **NEW-VIEW** messages have been delivered.

4.1.2. Problem to Resolve. Figure 4.2 illustrates the two failure modes that motivate Carry. In (a), a benignly slow leader misses the opportunity to certify its block before its view expires (*leader-slowness*); in (b), a Byzantine leader refuses to extend the previous correct leader’s block, preventing it from committing (*tail-forking*). Both modes leave a correct-leader proposal B_2 uncertified at the tail of the chain.

These issues are the core motivations for this work. The Carry mechanism that we introduce mitigates these issues while preserving the streamlined, efficient design of HS-like protocols.

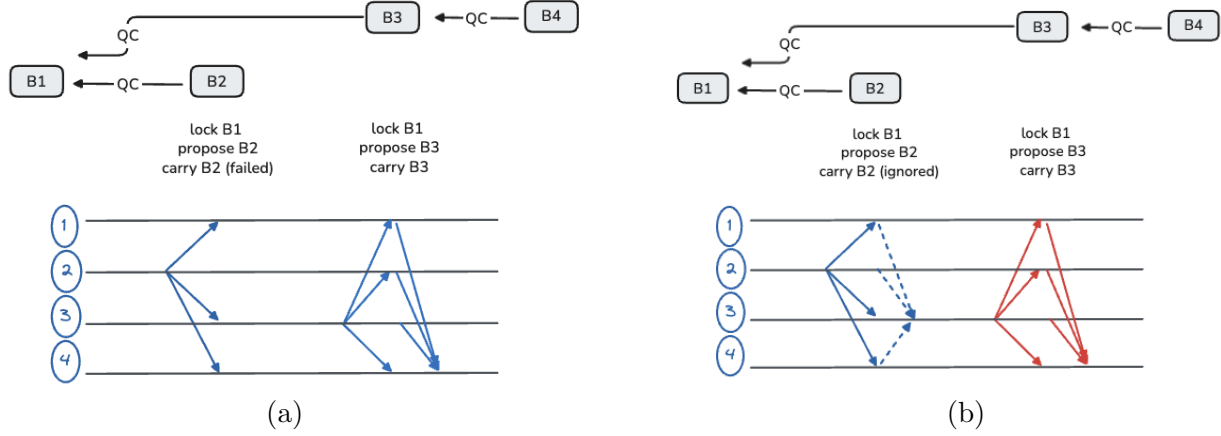


FIGURE 4.2. **Without tail protection:** (a) B_2 failed and benignly skipped. (b) B_2 tail-forked.

4.2. Carry

The Carry mechanism improves view synchronization and eliminates most leader-induced stalls. The core idea is straightforward: the next leader must *reinstall* the last uncertified block it sees, so that progress is not impinged on by the second QC's diffusion.

Recall that the core mechanism in HS-like protocols is a lock-commit regime: $2f+1$ replicas need to hold a lock on block B_x , proposed in view x , in order to commit it in the future. A replica acquires its lock on B_x precisely when it votes for a tail block T that extends B_x . For example, in HotStuff-2, which we use here to exemplify the Carry mechanism, T would directly extend B_x , i.e., $T.qc = QC(B_x)$. Although those votes are shipped in **NEW-VIEW** messages, nothing in HotStuff-2 requires the next leader to honor them; a sluggish leader may fail to collect $2f+1$ votes, and a Byzantine leader may ignore them, causing a *tail-forking* attack.

Core insight.

Votes are first-class citizens of the consensus process; replicas must guard their vote. Each vote implicitly endorses both the certified block and the extending block. Carry makes that endorsement *explicit*: replicas ship their latest vote (e.g., T) into the next view. The incoming leader proposal B_v must *reinstall* T , if needed, as a *reinstated block* $B_v.reinstated = T$. Thus, both the certified block *and* its immediate extension are protected if there are at least $f+1$ correct votes.

Carry Implementation. The key safety rules in Carry that are different from HotStuff-2 are defined below.

Algorithm 2 Carry-the-Tail Protocol (differences from HotStuff-2 in blue)

```
1: State: highest-vote and lock (the QC attached to highest-vote).
2:
3: Begin view  $v$ :
4: if replica is  $L_v \triangleright \text{leader}$  then
5:   wait until LEADERPROPOSALCONDITIONS( $v$ ); create  $B_v$ ; ATTACHMETADATA( $B_v$ ); propose
      $B_v$ 
6:
7: On receiving proposal  $B_v$ :
8: if VOTINGRULE( $B_v$ ) then
9:    $lock \leftarrow \text{view}(B_v.qc)$ ;  $highest\text{-}vote \leftarrow B_v$ 
10: Pacemaker.advanceView(); COMMITRULE( $B_v$ )
11:
12: On Pacemaker.exitView(): advance to view  $v$ ; send NEW-VIEW to next leader with (i) current
    view, (ii)  $lock$ , (iii) up to  $\rho$  vote-shares or empty-shares for each view  $w$  between  $lock$  and
    current view.
13:
14: VotingRule( $B_v$ ):  $\text{view}(lock) \leq \text{view}(B_v.qc)$  AND CARRYRULE( $B_v$ ).
15:
16: CarryRule( $B_v$ ):  $v > \text{view}(B_v.qc) + \rho$ , or (with  $T = highest\text{-}vote$ )  $B_v$  extends  $T$ ,  $T \succeq lock$ ,
    and  $B_v$  contains an empty-certificate or reinstated proposal for each view between  $\text{view}(B_v.qc)$ 
    and  $v$ .
17:
18: CommitRule( $B_v$ ): if  $B_v.qc = \text{QC}(B_{v-1})$  and  $B_{v-1}.qc = \text{QC}(B_{v-2})$ , commit the prefix up to
    (incl.)  $B_{v-2}$ .
19:
20: LeaderProposalConditions( $v$ ): (i) a fresh QC formed from  $2f+1$  NEW-VIEWS with identical
    highest votes for some  $w < v$ ; (ii)  $3f+1$  NEW-VIEWS received or Pacemaker.syncView().
21:
22: AttachMetadata( $B_v$ ):  $B_v.qc \leftarrow$  highest known QC.
23: if  $\text{view}(B_v.qc) + \rho \leq v \triangleright \text{eligible for reinstating}$  then
24:    $T \leftarrow$  highest known vote extending  $B_v.qc$ 
25:   if  $T \succ B_v.qc$  then
26:      $B_v.reinstated \leftarrow T$ 
27:   attach to  $B_v$  empty-certificates for each view between  $\text{view}(T)$  and  $v$ 
```

Carry Rule. A valid proposal B_v from the leader L_v of view v has the following format:

Highest QC:: the highest quorum certificate $\text{QC}(x)$ known to the leader L_v is attached as $B_v.qc = \text{QC}(x)$;

Reinstate:: if L_v received fewer than $2f+1$ votes for the highest tail T extending B_x , and $v-x \leq \rho$, then T is reinstated (in full) as $B_v.reinstated$;

Justification:: If $v-x \leq \rho$, then empty-certificates or reinstated proposals are attached to B_v for each view between x and v .

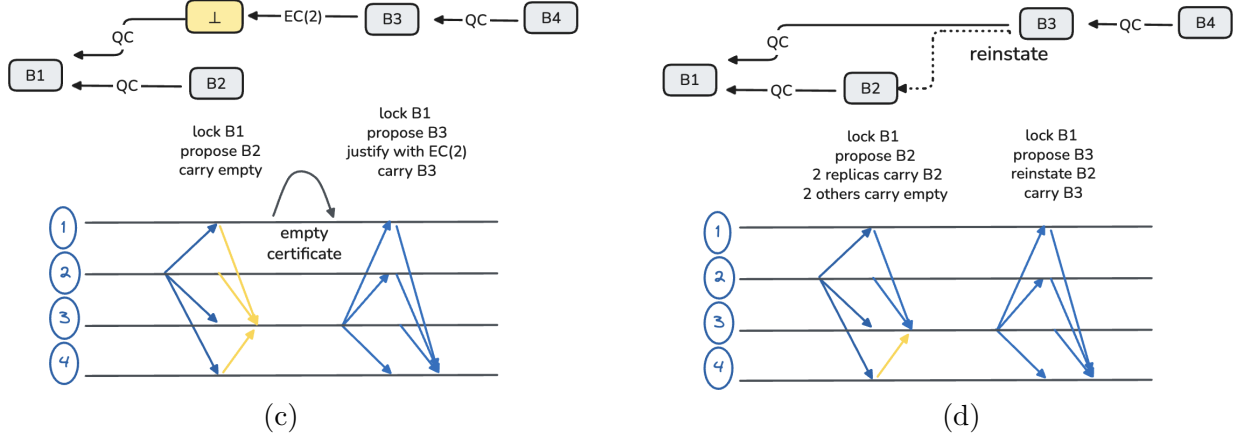


FIGURE 4.3. **With Carry protection:** (c) B_2 failed and EC-justified skipped. (d) B_2 reinstated.

Voting-Rule

A replica accepts the proposal B_v of leader L_v of view v if $B_v.qc$ has a higher or equal view than the replica's *lock*, and B_v adheres to the Carry Rule above.

Carry retains from HotStuff-2 the Commit Rule and Leader Proposal Rule.

4.2.1. The Carry-the-Tail Protocol. Akin to HotStuff-2, the Carry-the-Tail protocol flows view-by-view, as detailed in Algorithm 2. At the end of each view, the replicas and the incoming leader perform a handover protocol as follows:

Replica $\rightarrow$ **incoming leader.:** Each replica sends an incoming leader a **NEW-VIEW** message that carries

- (1) the next view number
- (2) the replica's highest QC (its *lock*)
- (3) its vote-shares (possibly empty) in the past ρ views.²

Leader $\rightarrow$ **replicas.:** On satisfying the Leader Proposal Rule, the leader of view v proposes a block B_v that

- (1) extends the highest QC it has collected, potentially freshly aggregated from **NEW-VIEW** messages, as $B_v.qc$,
- (2) reinstates T in full as $B_v.reinstated$, provided that the highest QC is from the past ρ views and the highest voted block T extends the highest QC,

²It is possible to send less information if the lock precedes the current view by less than ρ views; we omit this optimization for simplicity.

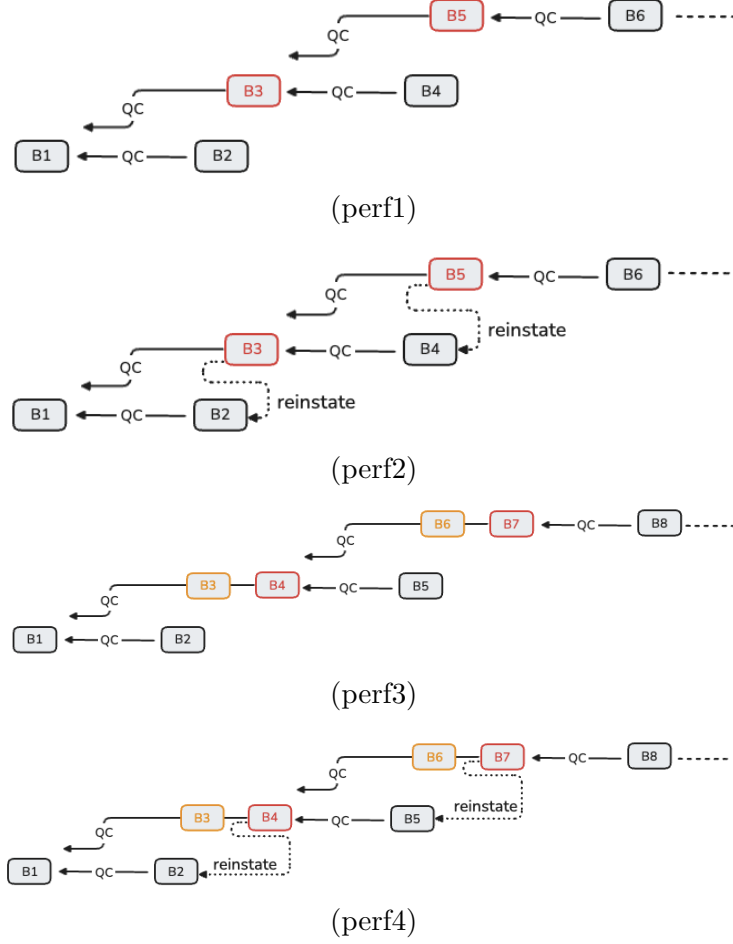


FIGURE 4.4. (perf1) Performance Degradation due to Tail-Forking. (perf2) Improvement with Reinstating. (perf3) Degradation under Two-Leader Attacks. (perf4) Improvement with ρ -Tail-Resilience.

- (3) attaches empty-certificates or reinstated proposals for each view between $\text{view}(B_v.qc)$ and v .

Figure 4.3 (c) depicts that a leader justifies its proposal with an *empty-certificate*; figure 4.3 (d) depicts that a leader reinstates a proposal with fewer than $2f + 1$ votes.

Note that if a leader receives two conflicting highest votes, it reinstates *both*; correct replicas will recognize the view as faulty and drop the conflicting votes. We omit the details from Algorithm 2 for brevity.

Correctness Proofs. We envision Carry as a drop-in mechanism for any HS-like protocol. In this chapter, we use HotStuff-2 to explain the design of Carry. Thus, Carry-the-Tail inherits the

safety and liveness properties of HotStuff-2, since the Carry mechanism only boosts performance but does not change the basic safety and liveness rules. We defer these proofs to Section 4.3 and concentrate here on proving the key novel property of Carry-the-Tail, ρ -tail-resilience.

THEOREM 4.2.1. *After GST, a tail block T proposed by a correct leader that is not ρ -isolated and receives $2\mathbf{f}+1$ votes will be extended by the next correct view.*

PROOF. As T is not ρ -isolated, there exists two correct views x and y , such that $x < \text{view}(T) < y \leq x + \rho$. We want to show that the correct leader L_y 's proposal B_y extends T in the chain. In particular, if B_y receives votes from all correct replicas before the view expires, then every correct replica locks on $B_y.qc$, and T will commit as soon as a later block in the chain is committed.

Easy Case: When $y = \text{view}(T)+1$, L_y forms a QC for T as the highest QC, proposes a block B_y that extends $\text{QC}(T)$ and disseminates it. All correct replicas become locked on $B_y.qc$, and T is guarded from ever being skipped.

Other Case: When $y > \text{view}(T) + 1$ and L_y obtains a QC for a view higher than $\text{view}(T)$. We need to show that B_y does not skip T , i.e., that $B_y.qc \succ T$. We do so by showing that between views x and y if a proposal P extends T (including T itself) and receives $\mathbf{f}+1$ correct votes, then P is extended by the next valid proposal P' . *Note:* since L_x is correct and $T.qc$ is from a view x or higher, the number of views between $T.qc$ and P' do not exceed ρ .

Assuming that P' exists and skips P . By the Carry Rule, P' must contains an *empty-certificate* for P . However, since P has $\mathbf{f}+1$ correct votes, it is impossible for justify the skipping with an empty certificate. P' must extends P .

From the argument above, there may be a succession of valid proposals between x and y extending T that are chained to each other via QCs or reinstated blocks. T is within this chain, and L_y terminates it. We obtain that $L_y \succ T$. $\square$

Vulnerability with $\rho + 1$ consecutive Byzantine leader.

After proving ρ -tail-resilience, we show that in Carry, ρ consecutive Byzantine leaders can still perform successful tail-forking attacks, as depicted in Figure 4.4(perf3), e.g., L_3 and L_4 are Byzantine and $\rho = 1$. First, L_{v+1} “fails” in view $v + 1$, not forming $\text{QC}(v)$. Next, B_{v+2} uses $\text{EC}(v+1)$ to justify skipping B_{v+1} . Furthermore, B_{v+2} skips B_v because no justification is required to skip B_v and no replica is locked on it.

To protect against tail-forking by two or more consecutive bad leaders, we apply the Carry method to the last ρ views. The NEW-VIEW messages from replicas should carry their votes, possibly empty, for the *last ρ views*. This protects a tail unless it is followed by ρ consecutive bad views. Hence, in each view rotation, all but (F_{actual}/ρ) proposals are protected from tail-forking. This property is referred to as ρ -Tail-Resilience. 2-Tail-Resilience is depicted in Figure 4.4(perf4). Importantly, for modest values of ρ , the chance of incurring ρ consecutive bad views in practice is very small, especially if leader rotation is randomized.

4.3. Correctness Proofs (continued)

First, we discuss the guarantees for non-equivocation and non-conflicting commitment.

LEMMA 4.3.0.1. *Let R_1 and R_2 be two correct replicas that lock certificates $QC(v)^1$ and $QC(v)^2$ of view v , respectively. Carry-the-Tail guarantees that $QC(v)^1 = QC(v)^2$.*

PROOF. a correct replica R_i locks a certificate $QC(v)^i$ if the Voting Rule is true, which can only happen if $QC(v)^i$ is attached to the proposal sent by the leader for view $v+1$ (or higher). Each of these QC's are composed of threshold signature-shares of $2\mathbf{f}+1$ replicas.

Let S_i be the replicas that voted for certificate $QC(v)^i$. Let $X_i = S_i - \mathbf{f}$ be the correct replicas in S_i . As $|S_i| = 2\mathbf{f}+1$, we have $|X_i| = 2\mathbf{f}+1 - \mathbf{f}$. If $QC(v)^1 \neq QC(v)^2$, then X_1 and X_2 must not overlap. Hence, $|X_1 \cup X_2| \geq 2(2\mathbf{f}+1 - \mathbf{f})$. This simplifies to $|X_1 \cup X_2| \geq 2\mathbf{f}+2$, which contradicts $\mathbf{n} = 3\mathbf{f}+1$. Hence, we conclude $QC(v)^1 = QC(v)^2$. $\square$

LEMMA 4.3.0.2. *If a replica R receives a certificate $QC(v+1)$ that extends certificate $QC(v)$, then no certificate $QC(w)$ conflicts with $QC(v)$, where view $w > v$, can exist.*

PROOF. If a replica R received $QC(v+1)$ that extends $QC(v)$, it implies that $2\mathbf{f}+1$ replicas that set $QC(v)$ as their highest lock agreed to vote for $QC(v+1)$. Let's denote the correct replicas in these $2\mathbf{f}+1$ replicas as A , who will not vote for a block conflicting with B_v .

We assume that $QC(w)$ is the lowest QC that conflicts with $QC(v)$ such that $w > v+1$. For $QC(w)$ to exist, there must be $2\mathbf{f}+1$ replicas who voted for it. Let's denote the correct replicas from these $2\mathbf{f}+1$ replicas as A' .

As A will not vote for B_w , thereby A and A' do not overlap. Then, $|A \cup A'| \geq 2(2\mathbf{f}+1 - \mathbf{f})$. This simplifies to $|A \cup A'| \geq 2\mathbf{f}+2$, which contradicts $\mathbf{n} = 3\mathbf{f}+1$. Hence, we conclude that $\text{QC}(w)$ could not exist. $\square$

COROLLARY 4.3.1. *If a correct replica R commits a block B_v , then no other conflicting block can commit.*

PROOF. Assume a block B_w , proposed in view w , conflicts with block B_v and another correct replica R' commits B_w . This implies that replicas R and R' have conflicting states. To commit B_v and B_w , R and R' must follow the Commit Rule, respectively: R must receive two consecutive QCs $\text{QC}(y+1)$ extending $\text{QC}(y)$ such that $y \geq x$, and R' must receive two consecutive QCs $\text{QC}(z+1)$ extending $\text{QC}(z)$ such that $z \geq w$. As B_v conflicts with B_w , then obviously $\text{QC}(y)$ conflicts with $\text{QC}(z)$. If we assume that $y > z+1$, from Lemma 4.3.0.2, we know that if $\text{QC}(z)$ and $\text{QC}(z+1)$ exist, then $\text{QC}(y)$ cannot exist, which contradicts the fact that R commits B_v ; If we assume that $z > y+1$, similarly, it contradicts the assumption that R' commits B_w . $\square$

Using these lemmas, next, we argue Carry-the-Tail's safety guarantees, which in blockchain systems ensure that at each entry in the global ledger (or log) there is a unique block.

THEOREM 4.3.1. *Carry-the-Tail guarantees consensus safety in a system with $\mathbf{n} \geq 3\mathbf{f}+1$ replicas: if two correct replicas R_1 and R_2 commit blocks A and B , respectively, at the same position k in the ledger, then $A = B$.*

PROOF. Lemma 4.3.0.1 helps to illustrate that within a view a leader cannot equivocate, that is, only one block can commit in a view. Furthermore, from Corollary 4.3.1, we know that two conflicting committed blocks cannot exist. This implies that both A and B are permanently part of the respective global ledgers of R_1 and R_2 and must not conflict.

Now, assume that $A \neq B$ and B extends A . Moreover, we know that A is at the position k in the ledger. Therefore, in the ledger, B should succeed A . As B is also at position k in the ledger, it implies that at least $k-1$ blocks precede B (including A), while at most $k-2$ blocks precede A . However, this contradicts the assumption that A is committed at the position k in the ledger with $k-1$ blocks preceding it. $\square$

Next, we discuss the liveness guarantee of the Carry-the-Tail protocol.

LEMMA 4.3.1.1. *After GST, a correct leader of view v can learn the highest QC across all correct replicas, all votes, and empty certificates for the past ρ views before v .*

PROOF. The View Synchronization mechanisms [42, 108] ensure that after GST the leader can receive NEW-VIEW messages from at least $2f+1$ correct replicas.

As each NEW-VIEW message from a replica R contains the *highest-QC* and *highest-vote* known to R and vote-shares of past ρ views before v . Thus, for past ρ views between the *highest-QC* and the current view v , the leader can either form empty certificates or learn votes. Also, the leader can learn the highest QC across all correct replicas or aggregate a higher one using collected votes. $\square$

LEMMA 4.3.1.2. *After GST, the B_v from a correct leader of view v will receive votes from all correct replicas.*

PROOF. The View Synchronization mechanisms ensure that after GST, each proposal arrives at all correct replicas before timeout. From Lemma 4.3.1.1, we know that each proposal contains the highest QC across all correct replicas, meeting the first part of the Voting Rule.

The second part checks the Carry Rule: if the QC is from a view that is more than ρ views earlier than v , then the Carry Rule is met. Otherwise, according to the protocol and Lemma 4.3.1.1, the proposal from a correct leader would contain (i) a reinstated block (highest-vote) extending the highest-QC, and (ii) all empty certificates between the reinstated block and view v , meeting the Carry Rule.

Thus, all correct replicas will satisfy the Voting Rule and will vote for the proposal. $\square$

THEOREM 4.3.2. *After GST, there is an unbounded number of committed blocks proposed by correct leaders.*

PROOF. Following Lemma 4.3.1.2, we can conclude that any sequence of three consecutive correct leaders L_v , L_{v+1} , and L_{v+2} will generate two consecutive QCs: $QC(v)$ and $QC(v+1)$, thereby committing block B_v and its prefix.

Given that the system comprises $n = 3f + 1$ replicas, such a trio of consecutive correct leaders is guaranteed to occur at least once in every sequence of n views. As the protocol proceeds, this ensures an unbounded number of block commitments. $\square$

4.4. Related Work

Rotational Leader. The historical perspective which leads to Carry-the-Tail stems from HotStuff [164], the first atomic broadcast protocol which incurs linear word-communication in steady state, even in the case of a handover from a faulty leader; and a quadratic word communication complexity in the worst case under a cascade of faulty leaders. RareSync [43] and Lewis-Pye [108] provide an implementation of HotStuff’s pacemaker module (that was left underspecified) that provides view-synchronization with worst-case quadratic communication cost and amortized linear.

The HS-like family of protocols reduces leader replacement communication costs to linear, enabling regular leader replacement at no additional communication cost or drop in system throughput. Additionally, these protocols streamline protocol phases to double the system throughput. Variants of HotStuff which maintain linearity include HotStuff-2 [118], which achieves two-phase latency while maintaining linearity; and HotStuff-1 [93], which contributes a streamlined variant of HotStuff-2 and speculative fast confirmation. Several other protocols have aimed two-phase streamlined and linear latency. However, Fast-HotStuff [89] and Jolteon [60] have quadratic complexity in view-change; AAR [8] employs expensive zero-knowledge proofs; Wendy [62] relies on a new aggregate signature construction (and it is super-linear); Marlin [154] introduces an additional *virtual block*, granting the leader an extra opportunity to propose a block supported by all correct replicas but falls back to 3-phase in the presence of failures.

Tail-Forking. BeeGees (“BG”) [65] indicated that HS-like protocols suffer significant degradation in throughput against the tail-forking problem (the original HotStuff variant may even lose liveness in an extreme scenario). Our work builds on the BG observation and borrows insights largely from it, while aiming to maintain linearity.

BG formulated a property called “*Any-Honest-Leader commit*” (AHL): after GST, once a correct leader proposes in a view, that block will be committed after at most k subsequent correct-leader views³. BG employs a complex leader handover in order to satisfy AHL. Rather than AHL, we formulate a property called ρ -tail-resilience. This property guarantees, under a reasonably small choice of ρ (e.g., $\rho = 6$), that a large constant fraction of correct leader proposals become committed even against worst-case tail-forking attacks. Note that this property is weaker than AHL, but suffices for the protocol to be censorship free and maintain effective good-put, while

³ k is a protocol parameter indicating the number of phases to reach a commit; typically, $k = 2$ or $k = 3$.

preserving linearity. In terms of technical ingredients, BG falls back into the regime set forth in PBFT [40]. This results in a complex leader handover regime: a view-change incurs quadratic word communication complexity and requires the leader to justify its proposal. A variant of BG replaces the explicit set of $2f + 1$ message with a SNARK, a complicated procedure with a high computational cost. We are somewhat concerned about the use of SNARKs to reduce word-communication complexity for two reasons: (i) throughput is bottle-necked by SNARK generation capacity, which is orders of magnitude lower than traditional BFT consensus throughput, and (ii) the original, uncompressed information is not guaranteed to be available.

The Carry mechanism aligns with the BG philosophy: use replicas' votes not merely to tally agreement, but to transport knowledge of the last safe block forward. However, there are two important differences.

- (1) Instead of leader justification, replicas guard their votes in Carry. They simply do not vote for a proposal by a future leader unless it extends their highest vote. This follows the same intuition as HotStuff, and thus preserves its two core tenets: linearity and simplicity.
- (2) The Carry mechanism allows leaders to reinstate a block even if they don't have a QC for it. This enhances good throughput against benign slowness, and a proposal by a slow leader can be committed even if it does not receive $2f+1$ votes.

Leader Slowness. The leader-slowness attack is a well-known problem in blockchains [48, 132, 139]. Prior work has illustrated that in Ethereum, for 59% of blocks, proposers have earned higher MEV rewards than block rewards [132], and any additional delay in proposing can help maximize their MEVs [144]. 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 [20]. (ii) Assign block rewards proportional to the number of attestations; a delayed block will receive fewer attestations and thus reduced block rewards [143]. However, if MEV rewards exceed total block rewards, the proposer makes a profit despite losing any block reward.

Linearity. Maintaining linearity of the HotStuff family of solutions is of paramount importance in this chapter. Hotstuff linearity has unlocked the first tight upper bounds to the atomic broadcast problem in a variety of settings which were open for decades.

Concretely, in pure asynchrony, VABA [6] provides the first tight upper bound for the Validated Byzantine Agreement problem. In particular, before HotStuff, the best known communication upper bound for this problem model was due to Cachin et al. [34], 2001, incurring $O(n^3)$ communication complexity in expectation. VABA harnesses HotStuff to arrive at a tight solution whose communication complexity is $O(n^2)$.

In the Authenticated Channels model (no signatures), Information Theoretic HotStuff [7] provides the first tight upper bound BA protocol with bounded communication complexity $O((f_a + 1) \cdot n^2)$ against f_a actual failures.

Real-World Deployments. In addition to foundational contributions, the defense provided by Carry against tail-forking may be valuable in practice for real life adoptions of HotStuff. At the time of this writing, blockchain companies which have announced employing some HotStuff variant at their core (to our knowledge) include Diem(Libra), Cypherium, Flow, Celo, Aptos, Espresso Systems, Pocket Network, SpaceComputer, and many others.

4.5. Conclusion

The Carry mechanism enhances HS-like protocols' robustness by protecting against tail-forking attacks. It also boosts performance under straggler leaders by ensuring safe progress with aggressive responsiveness without waiting for full quorums. Carry-the-Tail, a full solution that combines these methods with HotStuff-2, exhibits efficient linearity and high performance against both malicious leader and benign transient delays.

FairDAG: Consensus Fairness over Multi-Proposer Causal Design

Chapters 2–4 focused on the performance and safety of BFT consensus under the traditional *safety* and *liveness* goals. As Decentralized Finance develops, however, a new property has emerged as equally critical: the *fairness* of the final transaction order, because a Byzantine leader in control of ordering can extract Maximal Extractable Value (MEV) at the expense of others’ benefits. In this chapter, we ask: *can fair transaction ordering be guaranteed without being controlled by a single leader?* The material in this chapter appeared as [92].

Prior fair-ordering protocols and their limitation. Existing fair-ordering protocols [35, 45, 99, 100, 104, 124, 125, 169] address ordering manipulation by replacing the final ordering decided by a single leader with a final ordering *derived* from the local orderings collected from a quorum of replicas. POMPE [169] computes an *assigned ordering indicator* per transaction and guarantees *Ordering Linearizability* (if all correct replicas receives T_1 before any replica receives T_2 , then T_1 is ordered before T_2). THEMIS [99] builds a dependency graph from the collected local orderings and guarantees *γ -Batch-Order-Fairness* (if a γ -fraction of correct replicas receive T_1 before T_2 , then T_1 is ordered no later than T_2). Crucially, all of these protocols are built on top of a leader-based consensus layer [39, 164], and a Byzantine leader can selectively include only those local orderings that maximize its own MEV. Worse, broadcasting the collected local orderings has a quadratic communication complexity in the number of replicas at the leader, making the leader a throughput bottleneck even in the absence of Byzantine behavior.

Design goals. We argue that any well-designed fairness protocol should satisfy:

- G1 Resilience to Ordering Manipulation.** Limit Byzantine replicas’ influence on the final transaction order so that the targeted fairness property is preserved.
- G2 Minimal Correct-Replica Requirement.** Preserve fairness while relying on as few correct replicas as possible.
- G3 High Performance.** Minimize the overhead introduced by fair ordering, in both throughput and latency.

Leader-based fairness protocols inherit a structural conflict between G1, G2, and G3 from their consensus layer.

We propose FAIRDAG, a fairness framework that runs on top of *DAG-based* consensus protocols [18, 23, 49, 97, 149, 150]. Two features of DAG-based consensus directly address the goals above:

- **Multi-proposer high throughput.** Every replica proposes a block in every round¹ in parallel, removing the single-leader bottleneck (**G3**).
- **Validity through causal design.** Each DAG vertex references vertices from other replicas, forming a *Directed Acyclic Graph*. The resulting causal structure forces vertices proposed by correct replicas to be eventually committed by all correct replicas—constraining a Byzantine actor’s ability to selectively drop local orderings (**G1**) and lowering the required number of correct replicas (**G2**).

FAIRDAG comes in two variants: *absolute-ordering* FAIRDAG-AB, which guarantees Ordering Linearizability, and *relative-ordering* FAIRDAG-RL, which guarantees γ -Batch-Order-Fairness.

Two technical challenges arise when porting fair ordering onto a DAG-based layer. First, the higher commit latency of DAG protocols pushes us to use *uncommitted* vertices for fairness decisions, but different replicas may hold inconsistent views of those uncommitted vertices. Second, Byzantine replicas may try to manipulate the ordering by selectively ignoring vertices from specific replicas. FAIRDAG addresses both via a novel *Ordering Indicator Manager*, adaptive fairness thresholds, and new DAG-construction rules.

Contributions.

- (1) FAIRDAG-AB, an *absolute* fairness protocol guaranteeing Ordering Linearizability, built around an *Ordering Indicator Manager* and an adaptive fairness threshold (*LPAOI*) that is compatible with the multi-proposer design and the DAG commit rules.
- (2) FAIRDAG-RL, a *relative* fairness protocol guaranteeing γ -Batch-Order-Fairness; by leveraging the validity property of DAG-based consensus, we adopt new thresholds for the construction of dependency graphs that improve performance and reduce the minimal correct-replica requirement.

¹In DAG-based protocols, a *round* means the same as a *view*; we follow the DAG convention and use “round” in this chapter and Chapter 6.

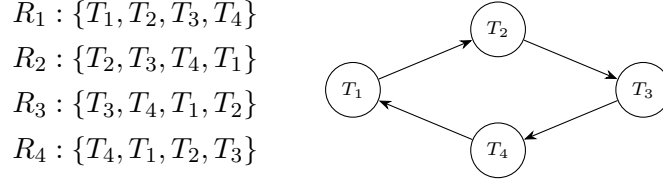


FIGURE 5.1. Condorcet Cycle

- (3) New DAG-construction rules that defend the resulting fair ordering against Byzantine replicas.
- (4) A comprehensive analytical and experimental evaluation showing that, against POMPE and THEMIS, both FAIRDAG-AB and FAIRDAG-RL achieve higher throughput and stronger fairness quality under both normal and adversarial conditions.

This chapter is structured as follows. Section 5.1 presents the background. Section 5.2 introduces the system model. Section 5.3 presents an overview of FAIRDAG. Section 5.4 and Section 5.5 describe FAIRDAG-AB and FAIRDAG-RL. Section 5.6 analytically compares FAIRDAG with POMPE and THEMIS. Section 5.7 presents the correctness proof. Section 5.8 presents the experimental evaluation. Section 5.9 discusses related work, and Section 5.10 concludes.

5.1. Background

FAIRDAG-AB and FAIRDAG-RL execute fairness protocols atop DAG-based consensus protocols. *Byzantine replicas*, corrupted by an adversary, may exhibit arbitrary malicious behavior, whereas the remaining *correct replicas* behave benignly. In this section, we present the definitions of three fairness properties and introduce the DAG-based consensus protocols.

5.1.1. Receive-Order-Fairness.

DEFINITION 5.1.1. *Receive-Order-Fairness.* *For any two transactions T_1 and T_2 , if all correct replicas receive T_1 before T_2 , then T_1 must be ordered before T_2 in the final ordering.*

We show that it is impossible to always guarantee *Receive-Order-Fairness* in the presence of Byzantine replicas for *Condorcet Cycles*. Figure 5.1 illustrates this impossibility with a concrete example. Consider four replicas R_1, R_2, R_3, R_4 , among which at most one may be Byzantine. Let there be four transactions T_1, T_2, T_3, T_4 . As shown on the left side of the figure, for any two commands T_i and T_{i+1} (modulo 4), three replicas receive T_i before T_{i+1} but the fourth differs.

Since the identity of the Byzantine replica is unknown, we respect all majority-endorsed orderings supported by at least three replicas when determining the final ordering. The right side of Figure 5.1 depicts a directed graph where an edge $T_i \rightarrow T_j$ indicates that at least three replicas received T_i before T_j . The resulting graph contains a Condorcet Cycle [31, 44]—a cycle of pairwise preferences that cannot be linearly extended without violating at least one of them. Hence, the *Receive-Order-Fairness* is impossible in this case.

5.1.2. Ordering Linearizability. *Ordering Linearizability* is a fairness property introduced by POMPE [169] and adopted by our protocol, FAIRDAG-AB. To achieve this property, each replica assigns a monotonically increasing *ordering indicator* (denoted oi) to each transaction reflecting the order it receives the transactions. The final ordering is then derived from ordering indicators collected from a majority of replicas.

Denoting by ois_i^C the set of ordering indicators for the transaction T_i from the correct replicas, we define *Ordering Linearizability*:

DEFINITION 5.1.2. *Ordering Linearizability.* *For any two transactions T_1 and T_2 , if all ordering indicators in ois_1^C are smaller than all those in ois_2^C , i.e.,*

$$\forall oi_1 \in ois_1^C, \forall oi_2 \in ois_2^C : oi_1 < oi_2.$$

then T_1 must be ordered before T_2 in the final ordering.

Figure 5.2 illustrates this definition with an example involving four replicas, where replicas R_1, R_2, R_3 are correct, and R_4 is Byzantine. Each replica assigns local ordering indicators to four transactions. For transactions T_1 and T_4 , the correct replicas assign $ois_1^C = \{2, 1, 1\}$ and $ois_4^C = \{3, 4, 4\}$, respectively. Since all indicators in ois_1^C are smaller than those in ois_4^C , any protocol satisfying Ordering Linearizability—such as POMPE and FAIRDAG-AB—must place T_1 before T_4 in the final ordering, regardless of the Byzantine replica’s input.

5.1.3. γ -Batch-Order-Fairness. In Section 5.1.1, we showed that it is impossible to guarantee *Receive-Order-Fairness* in the presence of unknown Byzantine replicas. However, a weaker yet practical variant, *γ -Batch-Order-Fairness*, can be achieved by both THEMIS [99] and our protocol FAIRDAG-RL.

$$\begin{aligned}
R_1 &: \{(T_2, 1), (T_1, 2), (T_4, 3), (T_3, 4)\} \\
R_2 &: \{(T_1, 1), (T_3, 2), (T_2, 3), (T_4, 4)\} \\
R_3 &: \{(T_1, 1), (T_2, 1), (T_3, 3), (T_4, 4)\} \\
R_4 &: \{(T_1, 1), (T_2, 2), (T_3, 3), (T_4, 4)\}
\end{aligned}$$

FIGURE 5.2. T_1 will be ordered before T_4 if *Ordering Linearizability* holds regardless of the local ordering from R_4 .

We say that a transaction T_2 is **dependent** on transaction T_1 , denoted as $T_1 \rightarrow T_2$, if T_1 must be ordered before T_2 in the final ordering. Due to the presence of *Condorcet cycles*, such dependencies can form cycles among transactions.

DEFINITION 5.1.3. A batch S of transactions is *cyclic dependent* if for any two transactions T_1, T_2 in S , there is a list of transactions that form a dependency path from T_1 to T_2 .

The final ordering can be partitioned into a sequence of non-overlapping batches $S_1, S_2, \dots$, where each S_i is a *maximal cyclically dependent batch*, i.e., for any i , $S_i \cup S_{i+1}$ is not cyclically dependent.

We say that T_1 is ordered **no later than** T_2 if T_1 is in the same or an earlier batch than T_2 . And we define *γ -Batch-Order-Fairness* as follows:

DEFINITION 5.1.4. **γ -Batch-Order-Fairness.** For any two transactions T_1 and T_2 , if at least a fraction γ of correct replicas receive T_1 before T_2 , then T_1 must be ordered no later than T_2 in the final ordering.

Figure 5.3 presents an example satisfying *γ -Batch-Order-Fairness*, where we have $\mathbf{n} = 4$ replicas (at most $\mathbf{f} = 1$ replica can be Byzantine) and 6 transactions. Assuming that the fairness protocol generates a final ordering that can be split into three *cyclic dependent batches*, for any two transactions, *γ -Batch-Order-Fairness* holds. For example, T_0 is received earlier than T_1 by correct replicas of $\gamma(\mathbf{n}-\mathbf{f}) = 3$ and T_0 is ordered in a batch earlier than T_1 .

5.1.4. DAG-based Consensus Protocols.

DAG-based BFT consensus protocols [49, 97, 150] operate in rounds². In each round r , every replica proposes a block, referred to as a DAG vertex. Each vertex references multiple vertices from the previous round $r-1$, represented as edges that encode the causal dependencies between DAG

²In DAG-based protocols and fair-ordering protocols, a *round* is a unit of progress, which serves the same role as a *view* as in Chapters 2-4; we follow the DAG convention and use “round” in this chapter and Chapter 6.

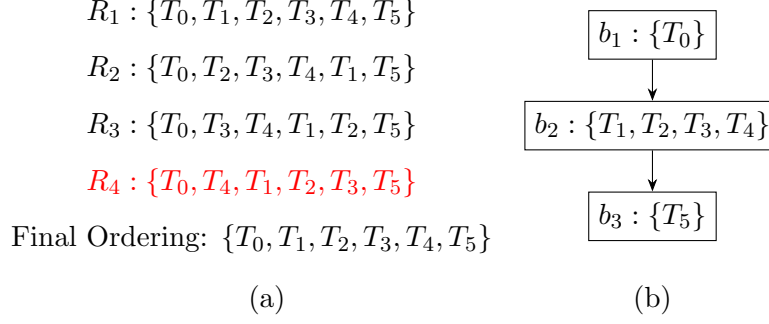


FIGURE 5.3. A final ordering of six transactions that satisfies γ -Batch-Order-Fairness with $\gamma = \frac{2}{3}$, 3 correct replicas, and 1 Byzantine replica.

vertices, forming a *Directed Acyclic Graph (DAG)*. The *causal history* of a DAG vertex includes all vertices reachable via the reference paths.

Every k rounds (e.g. $k = 2$ in TUSK [49]), a random or predetermined leader vertex is elected. These leader vertices are committed in an ascending order of round, and their causal histories are committed in a deterministic order. To ensure reliable dissemination, most DAG-based protocols employ reliable broadcast (RBC) mechanisms. With RBC and carefully designed commit rules, DAG-based protocols guarantee the following properties even in an *asynchronous network* without message delay bound:

- **Agreement:** If a correct replica commits a vertex v , then all correct replicas eventually commit v .
- **Total Order:** If a correct replica commits v before v' , then every correct replica commits v before v' .
- **Validity:** If a correct replica broadcasts a vertex v , then all correct replicas eventually commit v .

Compared to consensus protocols with a single leader, the multi-proposer design of DAG-based protocols enables higher throughput. But this comes at the cost of higher commit latency due to the overhead of RBC and multi-round commit rules.

5.2. System Model

We consider a distributed system consisting of a set of replicas and a potentially unbounded number of clients. The system is subject to Byzantine faults and operates under either asynchronous

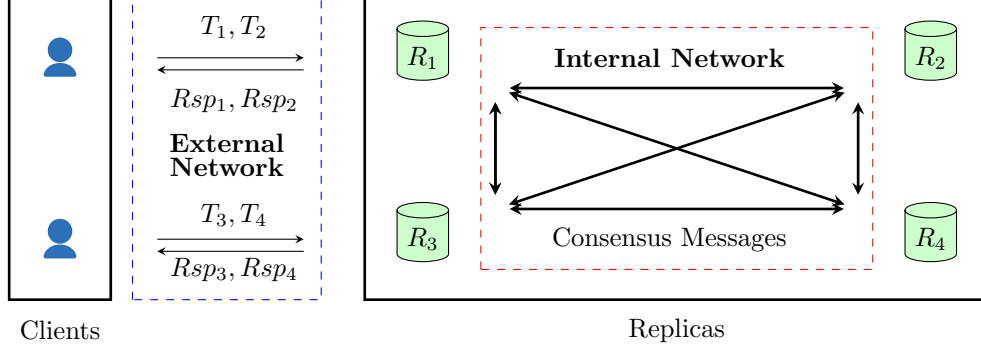


FIGURE 5.4. Network topology with Clients, Replicas, and Networks. Rsp_i represents a client response for transaction T_i , including the execution results of T_i .

or partially synchronous network conditions. Our model covers client behavior, replica corruption, authentication assumptions, and fairness-specific threat considerations.

5.2.1. Clients. Clients issue transactions to replicas and wait for execution results. They may behave arbitrarily with no correctness assumptions.

5.2.2. Replicas. In the system, there are a total of $\mathbf{n}$ replicas and an *adaptive adversary* capable of corrupting up to $\mathbf{f}$ replicas during execution. The corrupted replicas, referred to as Byzantine or malicious replicas, may exhibit arbitrary malicious behavior.

Regarding transaction ordering, Byzantine replicas may reorder transactions in local orderings and ignore unfavorable local orderings from other replicas. Correct replicas are honest about local orderings in which the transactions are received.

Fairness protocols differ in resilience to Byzantine faults. Specifically, FAIRDAG-AB and POMPE require $\mathbf{n} > 3\mathbf{f}$; THEMIS requires $\mathbf{n} > \frac{(2\gamma+2)\mathbf{f}}{2\gamma-1}$; and FAIRDAG-RL requires $\mathbf{n} > \frac{(2\gamma+1)\mathbf{f}}{2\gamma-1}$, $\frac{1}{2} < \gamma \leq 1$.

5.2.3. Authentication. We assume *authenticated communication*, where Byzantine replicas cannot forge messages from correct replicas. Authentication is enforced through *Public Key Infrastructure (PKI)* [95].

For integrity verification, each transaction T_i is associated with a digest d_i , computed using a *secure collision-resistant cryptographic hash function* [95].

5.2.4. Network. As illustrated in Figure 5.4, we distinguish between two types of networks: (1) *external communication* between clients and replicas, and (2) *internal communication* among replicas.

External network (client–replica). We make no assumptions about synchrony but assume the external network is *non-adversarial*. This assumption is necessary for preserving fairness, as an adversarial external network could arbitrarily control the order in which replicas receive transactions and then the final ordering.

Internal network (replica–replica). The internal network may operate either *asynchronously* or *partially synchronously*:

- **Asynchronous network:** Messages are never lost and are eventually delivered, but there is no bound on the message delays.
- **Partially synchronous network:** There exists an unknown *Global Stabilization Time* (*GST*) after which the message delays are bounded by a known constant Δ , i.e., any message sent at time t will be delivered by $\max(t, \text{GST}) + \Delta$.

5.3. Overview

Previous fairness protocols, such as POMPE [169] and THEMIS [99], are built atop leader-based consensus. Although the final ordering is derived from local orderings collected from a majority of the replicas, existing fairness protocols have these problems: (1) Byzantine leaders manipulate orderings and compromise fairness by selectively picking up the local orderings. (2) Byzantine or slow leaders can become system performance bottlenecks. To address these problems, FAIRDAG runs fairness protocols on top of multi-proposer DAG-based consensus with two layers:

- *DAG Layer* handles the dissemination (Figure 5.5(a,b)) and commitment (Figure 5.5(c)) of local orderings.
- *Fairness Layer* takes these committed local orderings (Figure 5.5(d)) as input to a transaction ordering algorithm (Figure 5.5(e)) to produce a final ordering (Figure 5.5(f)).

5.3.1. DAG Layer. The DAG layer adopts existing DAG-based consensus protocols—such as Tusk [49], DAG-Rider [97], and Bullshark [150]—with minimal modifications to support fairness guarantees.

Directed-Acyclic-Graph

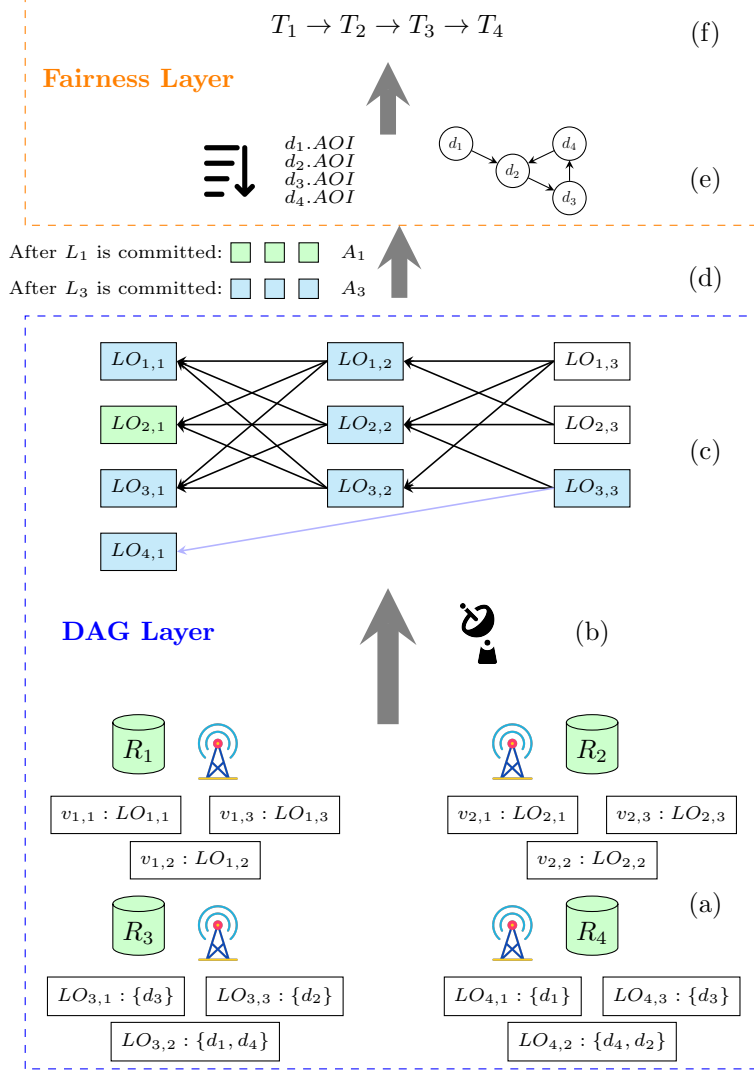


FIGURE 5.5. Architecture of FairDAG: (a) Replicas reliably broadcast blocks containing their local ordering fragments. (b) Each replica receives blocks delivered through Reliable Broadcast. (c) Each replica forms a local view of the DAG using received blocks and reference links, where different colors represent the subdags of different committed leader vertices. (d) Local orderings in subdag A_r are used as input of the fairness layer after L_r is committed. (e) Finalize transaction ordering using absolute ordering mechanism (left) or relative ordering mechanism (right), based on the committed local orderings. (f) A final transaction ordering is generated.

DAG-based protocols proceed in rounds where each replica concurrently proposes one vertex per round r . Each vertex references previous vertices, forming a DAG, where references are **edges**. Specifically, a vertex in round r references at least $\mathbf{n-f}$ vertices from round $r-1$ via **strong edges**, and optionally up to $\mathbf{f}$ vertices from earlier rounds via **weak edges**.

Let $v_{i,r}$ denote the vertex proposed by replica R_i in round r . A valid vertex satisfies:

- *strong_edges*: includes at least $\mathbf{n} - \mathbf{f}$ *strong edges*,
- *weak_edges*: includes up to $\mathbf{f}$ *weak edges*.

To guarantee the fairness properties in the presence of adversary, we extend existing DAG-based protocols by applying new rules of forming DAG vertices:

- (1) In FAIRDAG, replicas include weak edges in vertices to incorporate local orderings from slower replicas, ensuring that all correct local orderings eventually get committed.
- (2) To preserve the integrity of a replica's local ordering, each vertex $v_{i,r}$ must include a strong edge to its vertex in the previous round $v_{i,r-1}$ (for $r > 0$).

Figure 5.5(c) shows an DAG example where black arrows represent strong edges, and blue arrows represent weak edges.

Reliably broadcasting a vertex

To address issue (1), in FAIRDAG, instead of relying on a leader to collect local orderings, each replica R_i autonomously reliably broadcast [29, 36, 49, 67] its local ordering LO_i . As Figure 5.5(a) shows, FAIRDAG vertex $v_{i,r}$ contains r -th slice of the replica R_i 's *local ordering* ($LO_{i,r}$)—a sequence reflecting the order in which transactions were received by replica R_i . Each local ordering is represented as a sequence of transaction digests paired with monotonically increasing ordering indicators.

Constructing a DAG

As illustrated in Figure 5.5 (a,b), the DAG layer of FAIRDAG enables replicas to reliably broadcast and deliver vertices from one another. This multi-proposer design alleviates the performance bottlenecks associated with a single leader, thereby addressing issue (2).

Using the delivered vertices and the references within them, each replica constructs its local view of the DAG, as shown in Figure 5.5(c). As mentioned in Section 5.1.4, the DAG protocols guarantee that all correct replicas eventually have consistent DAG views.

Committing DAG vertices

The commit rules in DAG-based consensus protocols—leveraging the causal dependencies between vertices—further mitigate leader manipulation in selecting local orderings, addressing issue (1).

DAG protocols group rounds into *waves*, each containing one or more leader vertices. A leader vertex L_{r_i} in round r_i is *committed* once specific conditions are met. Let C_{r_i} denote its *causal history*—the set of vertices reachable from L_{r_i} , including L_{r_i} itself. DAG protocols ensure that all correct replicas commit leader vertices in a consistent, round-increasing order $(L_{r_1}, L_{r_2}, \dots)$ such that $C_{r_i} \subset C_{r_{i+1}}$ for all i .

A vertex v is said to be in the subdag of leader L_{r_i} if $v \in C_{r_i}$ and $v \notin C_{r_j}$ for all $j < i$. Let A_{r_i} denote the vertices in the subdag of L_{r_i} . The committed DAG can thus be partitioned into non-overlapping subdags $(A_{r_1}, A_{r_2}, \dots)$, each corresponding to a committed leader.

For example, in Figure 5.5(c), L_1 is the green vertex $v_{2,1}$ and L_3 is the blue vertex $v_{3,3}$. Then A_1 consists of the green vertices, and A_3 consists of the blue ones.

As shown in Figure 5.5(d), each time a leader L_r is committed, the fairness layer processes A_r to compute the final ordering.

The DAG layer satisfies a necessary condition to guarantee that final ordering aligns with the preferences of the majority of correct replicas, as each subdag A_r aggregates local orderings from at least $\mathbf{n-f}$ replicas—guaranteeing the inclusion of at least $\mathbf{n-2f}$ correct replicas. (Note: $\mathbf{n-2f} \geq \frac{\mathbf{n-f}}{2}$ holds for all protocols.)

5.3.2. Fairness Layer. Taking local orderings within the committed subdags as input, the fairness layer runs a transaction ordering algorithm to calculate the final transaction ordering.

In **absolute** fairness protocols (Figure 5.5(e), left), each transaction gets an *assigned ordering indicator (AOI)* based on the input local orderings. The final ordering (Figure 5.5(f)) is then derived, sorting transactions by their AOI. FAIRDAG-AB is an *absolute* fairness protocols and satisfies the *Ordering Linearizability* property.

In **relative** fairness protocols (Figure 5.5(e), right) construct a dependency graph of transaction nodes, where edges encode pairwise ordering dependencies derived from the input local orderings. The final ordering is a *Hamiltonian path* within the graph (Figure 5.5(f)). FAIRDAG-RL is an *relative* fairness protocol and satisfies the γ -Batch-Order-Fairness property.

5.4. FairDAG-AB

FAIRDAG-AB is an **absolute** fairness protocol. In this section, we demonstrate how FAIRDAG-AB calculates the *assigned ordering indicator (AOI)* for each transaction and orders the transactions based on the AOI values.

5.4.1. Transaction Dissemination. To prevent leader-driven manipulation, clients broadcast transactions to all replicas. This prevents adversarial replicas from censoring transactions to influence their position in final ordering.

Upon receiving a client transaction T , replica R gives T a monotonically increasing ordering indicator oi from its local timer, appends digest d of T and oi to its lists dgs and ois (Figure 5.6, Line 2), respectively, which are in-memory variables storing transaction digests and ordering indicators of pending transactions (Lines 7-10).

5.4.2. FairDAG-AB Vertex. Built atop a DAG-based consensus protocol, each FAIRDAG-AB replica constructs and reliably broadcasts a DAG vertex when protocol conditions are met. In addition to DAG-specific metadata (as described in Section 5.3.1), a FAIRDAG-AB vertex includes (and clears) the replica’s local ordering of pending transactions it received after it broadcast the last vertex, encoded in dgs , ois (Lines 11-15).

5.4.3. Managing and Assigning Ordering Indicators. FAIRDAG-AB leverages local orderings within committed DAG vertices to determine the final transaction ordering. However, a delay exists between receiving and committing the DAG vertices. To efficiently manage and utilize local orderings from both committed and uncommitted vertices, FAIRDAG-AB maintains an *Ordering Indicator Manager* ($OIM(d)$) for each transaction digest d , which tracks the replica’s local view of the DAG and ordering indicators within the vertices.

We say that a replica R has **seen** an ordering indicator oi if oi is in a vertex in R ’s local DAG view. R has **committed** oi if oi is in a committed vertex. $OIM(d)$ contains the following information (Lines 4-6):

- *seen_ois*: ordering indicators of d that R has seen.
- *committed_ois*: ordering indicators of d that R has committed.
- *LPAOI*: the lowest possible value of the assigned ordering indicator of d .
- *AOI*: the value of the assigned ordering indicator of d .

```

1: State Variables (per replica)
2:  $txns\_w\_assigned\_oi := \{\}$ ;  $highest\_oi\_list[1..n] := 0$ ;  $dgs, ois := []$  ;
3:  $current\_round, replica\_id$ 

4: Ordering Indicator Manager (OIM)
5:  $seen\_ois[1..n] := \infty$ ,  $committed\_ois[1..n] := \infty$ 
6:  $LPAOI := \infty$ ,  $AOI := \infty$ 

   Client Thread (processing client transactions) :
7: event On receive transaction  $T$  do
8:   if  $T$  is valid then
9:      $oi := local\_timer$ 
10:     $dgs.append(T.digest)$ ;  $ois.append(oi)$ 

   DAG Layer Thread (constructing the DAG) :
11: event On propose DAG vertex do
12:    $v := DAGVertex(replica\_id, current\_round)$ 
13:   Add pending local ordering  $dgs$  and  $ois$  into  $v$ 
14:   Clear  $dgs$  and  $ois$ 
15:   Reliably broadcast  $v$ 

16: event On deliver  $v_{i,r}$  do
17:    $highest\_oi\_list[i] :=$  the highest  $oi$  in  $v_{i,r}.ois$ 
18:   for  $(d, oi) \in (v_{i,r}.dgs, v_{i,r}.ois)$  do
19:      $OIM(d).seen\_ois[i] := oi$ 

20: event On commit  $L_r$  do
21:   Send  $A_r$  to Fairness Layer

   Fairness Layer Thread (Ordering transactions) :
22: event On receive  $A_r$  do
23:   for  $v \in A_r$ ,  $(d, oi) \in (v.dgs, v.ois)$  do
24:      $OIM(d).committed\_ois[v.replica\_id] := oi$   $\triangleright$  Update committed ois
25:    $LPAOI_{min} := \infty$ 
26:   for  $d$  such that  $OIM(d) = \infty$  do
27:     if  $OIM(d)$  has at least  $n - f$  committed ois then
28:        $OIM(d).AOI := (f+1)$ -th smallest in  $committed\_ois$   $\triangleright$  Calculate AOI
29:       Add  $d$  to  $txns\_w\_assigned\_oi$ 
30:     else
31:       for  $i = 1$  to  $n$  do
32:          $lp\_ois[i] := \min(d.seen\_ois[i], highest\_oi\_list[i])$ 
33:          $d.LPAOI := (f+1)$ -th smallest in  $lp\_ois$   $\triangleright$  Calculate LPAOI
34:          $LPAOI_{min} := \min(LPAOI_{min}, d.LPAOI)$   $\triangleright$  Track the minimal LPAOI of all transactions without an AOI
35:   sort  $txns\_w\_assigned\_oi$  sorted by  $AOI$ 
36:   execute all transactions with an  $AOI$  lower than  $LPAOI_{min}$ 

```

FIGURE 5.6. FairDAG-AB Algorithm.

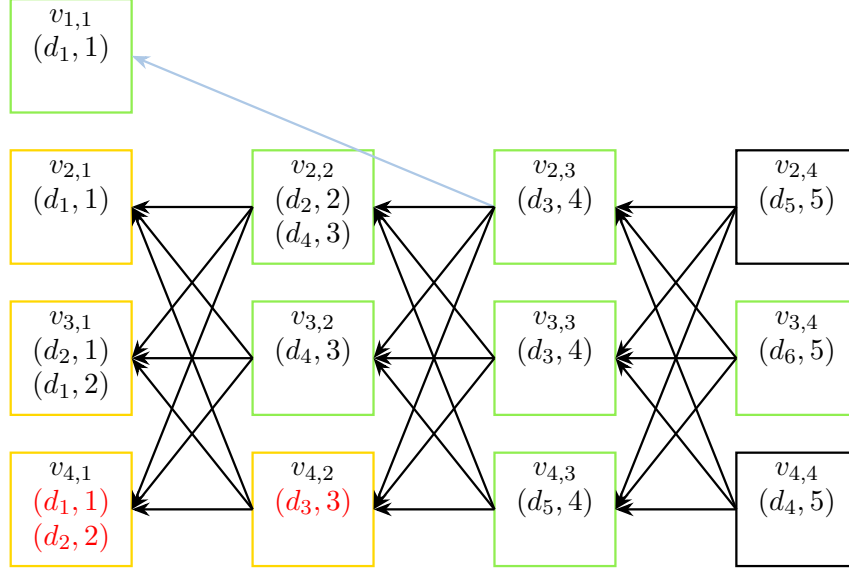


FIGURE 5.7. Example of calculating AOI and LPAOI.

The final ordering is determined using *AOI* values derived from *committed_ois*. *LPAOI*, computed from *seen_ois*, helps determine when it is safe to decide the position of a transaction in the final ordering (see Section 5.4.4). The *seen_ois* and *committed_ois* are indexed from 1 to $\mathbf{n}$ and each item is initialized to ∞ . For example, if $OIM(d).seen_ois[2] = 3$, it means that R has seen an ordering indicator 3 for the transaction with digest d from replica R_2 .

Managing Ordering Indicators

Besides the *OIM* of each transaction digest d , each FAIRDAG-AB replica R also maintains *highest_ois*: a vector of the highest ordering indicators received from each replica, initialized to be 0 and indexed from 1 to $\mathbf{n}$ (Line 2). This vector is essential for checking whether it is safe to determine the position of a transaction in final ordering (See details in Section 5.4.4).

Upon a DAG vertex $v_{i,r}$ is delivered and added to replica R 's local DAG view, R updates *highest_ois*[i] to the highest ordering indicator in $v_{i,r}$. For each digest d in $v_{i,r}$, R updates $OIM(d).seen_ois[i]$ (Lines 16-19).

Upon a DAG leader vertex L_r is committed, its subdag A_r is input to the fairness layer (Lines 20-21). And R updates *committed_ois* for the digests in A_r accordingly (Lines 22-24).

Calculating Assigned Ordering Indicator

If $OIM(d)$ has at least $\mathbf{n}-\mathbf{f}$ non- ∞ *committed_oi* values R calculates $OIM(d).AOI$. The *AOI* is the $(\mathbf{f}+1)$ -th smallest value of the *committed_ois*, which definitely falls within the range

of ordering indicators from correct replicas. Then, d is added to $txns_w_assigned_oi$, which is set of transactions digests with valid AOI values (Lines 26-29). *Note: The value of AOI is immutable once calculated, even if more ordering indicators of d are committed.*

EXAMPLE 5.4.1. Figure 5.7 illustrates how to calculate AOI values. Suppose $v_{4,2}$ and $v_{3,4}$ are committed leader vertices L_2 and L_4 , with yellow A_2 and green A_4 , respectively. In A_2 , digest d_1 has committed ordering indicators $\infty, 1, 2, 1$. Since this includes at least $\mathbf{n}-\mathbf{f}$ valid values, its AOI is the $(\mathbf{f}+1)$ -th lowest, which is 1 (Lines 26-29). Although an additional $seen_ois$ of 1 appears in $v_{1,1} \in A_4$, it is ignored since AOI is immutable once calculated after L_2 is committed. In A_4 , digest d_2 has committed ordering indicators $\infty, 1, 2, 3$, yielding an AOI of 2. Similarly, d_3 receives an AOI of 3. Other digests in A_4 lack sufficient committed ois data for AOI .

5.4.4. Global Ordering.

Due to the randomness in message arrival times, DAG vertices with smaller local ordering indicators may be committed later than vertices from other replicas with higher ordering indicators. As a result, a transaction d_1 may get a smaller AOI than d_2 , even if d_1 obtains its AOI in a later round than d_2 . To ensure strict ordering of transactions based on their AOI values—which is essential for achieving *Ordering Linearizability*—it is necessary to guarantee that no other transaction could get a lower AOI before determining the position of a transaction in the final ordering. To enforce it, for each transaction without an AOI , we calculate its $LPAOI$, the lowest AOI value it could possibly get. And we track $LPAOI_{min}$, the minimal $LPAOI$ of transactions without an AOI .

Calculating $LPAOI$

For each transaction digest d without a valid AOI , replica R constructs a vector lp_ois , where each entry is computed as $lp_ois[i] = \min(OIM(d).seen_ois[i], highest_ois[i])$. The $LPAOI$ of d is then defined as the $(\mathbf{f}+1)$ -th smallest value in lp_ois . And $LPAOI_{min}$ is the minimal value across all $LPAOI$ values (Lines 30-34).

$LPAOI_{min}$ serves as a threshold: only transactions with $AOI < LPAOI_{min}$ can determine its position in final ordering. The fairness layer sorts all digests in $txns_w_assigned_oi$ by AOI , executes all transactions with an AOI lower than the threshold. The ordered digests are then removed from $txns_w_assigned_oi$ (Lines 35-36).

EXAMPLE 5.4.2. As shown in Figure 5.7, transaction digests d_1 , d_2 , and d_3 have AOI values of 1, 2, and 4, respectively. Given $highest_ois = (2, 6, 6, 6)$ and $OIM(d_4).seen_ois = (\infty, 3, 3, 5)$, digest d_4 derives its $lp_ois = (2, 3, 3, 5)$ and obtains $LPAOI = 3$. Similarly, d_5 and d_6 have $LPAOI$ values of 4 and 5. $LPAOI_{\min}$ is then 3, (Lines 30-34) so d_1 and d_2 can be ordered and executed. In contrast, d_3 cannot be executed since its AOI exceeds $LPAOI_{\min}$, implying that d_4 could possibly get a lower AOI than d_3 .

5.5. FairDAG-RL

FAIRDAG-RL is a **relative** fairness protocol. In this section, we demonstrate how to construct dependency graphs between transactions and derive a final transaction ordering from the dependency graphs.

5.5.1. Transaction Dissemination. As in FAIRDAG-AB, clients broadcast transactions to all replicas for censorship resistance. Upon meeting the conditions in DAG protocols, each replica broadcasts a vertex. A FAIRDAG-RL vertex is a restricted form of a FAIRDAG-AB vertex: it includes a sequence of incrementing counter values as ordering indicators corresponding to received transactions (Figure 5.8 Lines 7-8). For simplicity, we omit the ordering indicators in Figure 5.9.

The local orderings in subdag A_r will be input to the fairness layer after L_r is committed in FAIRDAG-RL (Lines 14-15).

5.5.2. Dependency Graph Construction. The fairness layer of FAIRDAG-RL utilizes local orderings in committed DAG vertices to construct dependency graphs that reflect the ordering preferences of replicas. Every time a leader vertex is committed, a new dependency graph is constructed:

First, transaction digests from A_r are added as graph nodes if it has not been added to any dependency graph is previously rounds. Each transaction digest d is associated with a node that stores:

- *type*: the type of the transaction node (See details later in **Adding nodes**)
- *committed_ois*: a vector of committed ordering indicators for d .
- *committed_rounds*: the rounds in which the corresponding *committed_oi* is committed.
- G : the graph to which the node is added.

```

1: State Variables
2: graphs := []
3: current_round, replica_id
4: counter := 0

   Client Thread (processing transactions from clients) :
5: event On receive a transaction T do
6:   if T is valid then
7:     counter := counter + 1
8:     ois.append(counter), dgs.append(T.digest)

   DAG Layer Thread (forming the DAG) :
9: event On propose vertex do
10:  v := DAGVertex(replica_id, current_round)
11:  v.dgs := dgs, v.ois := ois
12:  dgs.clear(), ois.clear()
13:  Reliably broadcast v

14: event On commit(Lr) do
15:  Send Ar to Ordering Layer

```

FIGURE 5.8. FairDAG-RL: Transaction dissemination and DAG vertex proposal.

Second, pairwise ordering preferences are aggregated into a weight function: $weight(d_1, d_2)$ denotes the number of replicas who have a lower *committed_{oi}* for d_1 than d_2 . Each dependency graph contains the following information:

- *nodes*: the set of transaction digest nodes.
- *weight*: mapping of the pairs of nodes to their weights.
- *edges*: directed edges representing inferred ordering constraints between the transaction digest nodes.

Third, a directed edge from d_1 to d_2 is added if $weight(d_1, d_2)$ exceeds a quorum-based threshold $\frac{n-f}{2}$ and is not lower than $weight(d_2, d_1)$.

Thresholds

To construct a dependency graph that reflects the ordering preferences of the majority and mitigates manipulation by Byzantine replicas, relative fairness protocols should leverage as many correct local orderings as possible. Prior protocols such as THEMIS [99] and RASHNU [125] rely on a single leader to collect local orderings, where a Byzantine leader may intentionally exclude local orderings from up to f correct replicas. As a result, a transaction may appear in at most $n-2f$ committed local orderings if Byzantine replicas ignore the transaction and the Byzantine leader

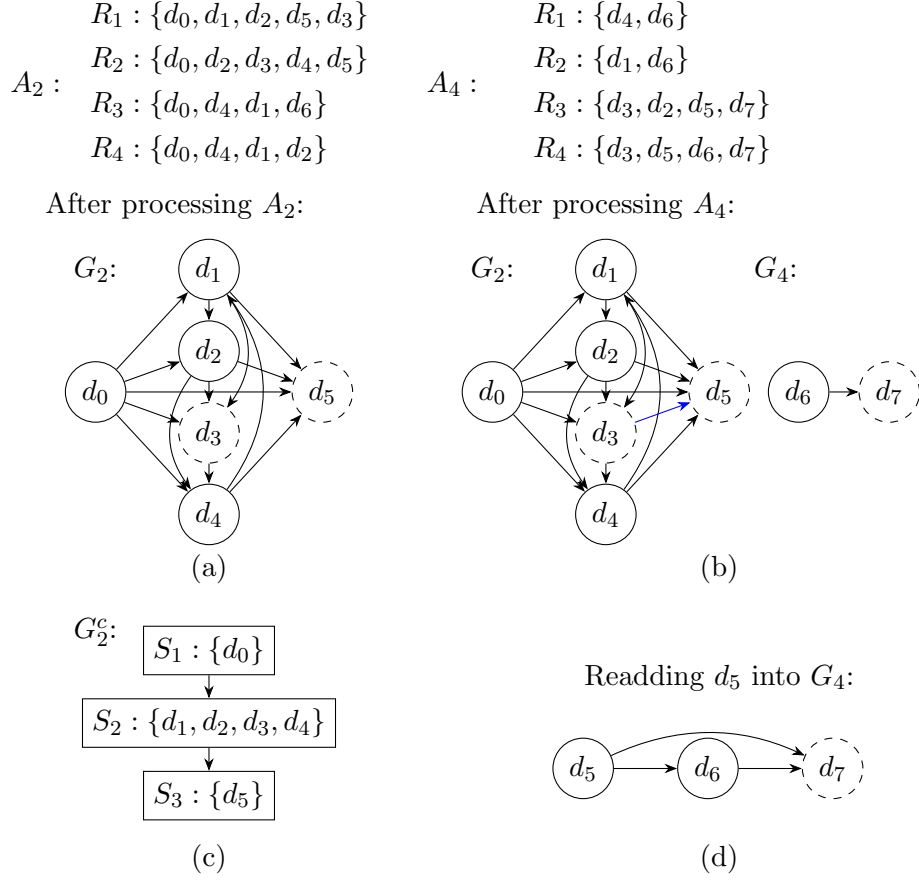


FIGURE 5.9. Constructing dependency graphs and finalizing transaction order with $\mathbf{n} = 4, \gamma = 1, \mathbf{f} = 1$.

excludes $\mathbf{f}$ correct local orderings, increasing both the susceptibility to ordering manipulation and the minimal requirement of number of correct replicas (see Section 5.6 for details).

In contrast, DAG-based consensus protocols ensure that all correct local orderings are eventually committed. This guarantees that each transaction can appear in at least $\mathbf{n} - \mathbf{f}$ committed local orderings. Consequently, FAIRDAG-AB raises the thresholds used in the construction of dependency graphs from $\mathbf{n} - 2\mathbf{f}$ and $\frac{\mathbf{n}-2\mathbf{f}}{2}$ (used in THEMIS and RASHNU) to $\mathbf{n} - \mathbf{f}$ and $\frac{\mathbf{n}-\mathbf{f}}{2}$, respectively. We will elaborate on the details below.

Adding nodes

For each transaction digest d in A_r , let $node(d)$ represent its dependency graph node. Its *committed_ois* and *committed_rounds* are updated using information contained in A_r , and the

Fairness Layer Thread (Ordering Transactions) :

```

1: event On receive  $A_r$  do
2:    $G_r := \text{NewGraph}()$ ,  $\text{graphs.push}(G_r)$ 
3:   ▷Find nodes updated with  $A_r$ 
4:    $\text{updated\_nodes} := \{\}$ 
5:   for  $v \in A_r$  do
6:      $i := v.\text{replica\_id}$ 
7:     for  $(d, oi) \in (v.dgs, v.ois)$  do
8:        $\text{node}(d).\text{committed\_ois}[i] := oi$ 
9:        $\text{node}(d).\text{committed\_rounds}[i] := r$ 
10:       $\text{updated\_nodes.insert}(d)$ 
11:   ▷Update node types and add new non-blank nodes to  $G_r$ 
12:   for  $d \in \text{updated\_nodes}$  do
13:     if  $\text{node}(d).\text{type} = \text{blank}$  then
14:       Let  $ap(d, r)$  be the number of ordering indicators for  $d$  that are committed by round  $r$ 
15:       if  $ap(d, r) \geq n - f$  then
16:          $\text{node}(d).\text{type} := \text{solid}$ ;  $G_r.\text{nodes.add}(\text{node}(d))$ 
17:       else if  $ap(d, r) \geq \frac{n-f}{2}$  then
18:          $\text{node}(d).\text{type} := \text{shaded}$ ;  $G_r.\text{nodes.add}(\text{node}(d))$ 
19:   ▷Find all candidate edges in all existing graphs  $G_r$ 
20:    $\text{addable\_edges} := \{\}$ 
21:   for  $v \in A_r$  do
22:      $i := v.\text{replica\_id}$ 
23:     for  $(d, oi) \in (v.dgs, v.ois)$  do
24:        $G' := \text{node}(d).G$ 
25:        $d\_oi := \text{node}(d).\text{committed\_ois}[i]$ 
26:       for  $\text{node}(d_2) \in G'.\text{nodes}$  do
27:         if  $d\_oi < \text{node}(d_2).\text{committed\_ois}[i]$  then
28:           increment  $G'.\text{weight}[(d, d_2)]$ 
29:         else
30:           increment  $G'.\text{weight}[(d_2, d)]$ 
31:         if either weight reaches threshold  $\frac{n-f}{2}$  then
32:            $\text{addable\_edges.insert}(d, d_2)$ 
33:   ▷Add edge if weight reaches threshold
34:   for  $(d, d_2) \in \text{addable\_edges}$  do
35:      $G := \text{node}(d).G$ 
36:     if  $G.\text{weight}[(d, d_2)] \geq G.\text{weight}[(d_2, d)]$  then
37:        $G.\text{edges.add}(e(d, d_2))$ 
38:     else
39:        $G.\text{edges.add}(e(d_2, d))$ 
40:   ▷Finalize Transaction Ordering in  $G_r$  when  $G_r$  is a tournament
41:    $\text{ORDERFINALIZATION}()$ 

```

FIGURE 5.10. Dependency graph construction in FairDAG-RL.

node is added to *updated_nodes*, a set of transactions whose *committed_ois* are updated because of A_r (Figure 5.10 Lines 3-10) ³.

³All operations related to dependency graph construction only apply to transactions that are not ordered yet. We omit this for simplicity in the protocol description and the pseudocode.

Then, we check if any transaction digests can be added as nodes into the new dependency graph of round r . We define $ap(d, r)$ as the number of ordering indicators for d that are committed by round r :

$$ap(d, r) := |\{i | node(d).committed_rounds[i] \leq r\}|$$

Each node in *updated_nodes* whose type is *blank*, which means that it has not been added into any dependency graph previously, is classified as:

- *solid*, if $ap(d, r) \geq \mathbf{n} - \mathbf{f}$.
- *shaded*, if $\frac{\mathbf{n}-\mathbf{f}}{2} \leq ap(d, r) < \mathbf{n} - \mathbf{f}$.
- *blank*, if $ap(d, r) < \frac{\mathbf{n}-\mathbf{f}}{2}$.

Then we add the *non-blank* nodes to G_r (Lines 11-18). Since classification is only applied to previously *blank* nodes, it is guaranteed that each digest is inserted into at most one dependency graph.

Updating weights between nodes

After classifying and adding nodes to G_r , we update edge weights based on local orderings in A_r . Vertices in A_r are processed in round-increasing order.

For each vertex v from replica R_i , we iterate through the transaction digests in $v.dgs$. For each digest d , we compare the value of $node(d).committed_ois[i]$ with all other nodes in the graph G of $node(d)$. For each pair (d, d_2) , we increment $G.weight[(d, d_2)]$ if $node(d).committed_ois[i] < node(d_2).committed_ois[i]$; otherwise, we increment $G.weight[(d_2, d)]$ (Lines 21-30).

During this process, we maintain a set *addable_edges* to track the edges that can be added to the dependency graph. If $G.weight[(d, d_2)]$ or $G.weight[(d_2, d)]$ reaches the threshold $\frac{\mathbf{n}-\mathbf{f}}{2}$, the corresponding pair is added to *addable_edges* (Lines 31-32).

Adding edges

For each pair $(d, d_2) \in addable_edges$, if there is no edge between $node(d)$ and $node(d_2)$, an edge is added based on the majority preference (Lines 33-39):

- If $G.weight[(d, d_2)] \geq G.weight[(d_2, d)]$, add edge $e(d, d_2)$ from $node(d)$ to $node(d_2)$;
- Otherwise, add edge $e(d_2, d)$ from $node(d_2)$ to $node(d)$.

EXAMPLE 5.5.1. *Figure 5.9 illustrates how FAIRDAG-RL constructs dependency graphs. In Figure 5.9(a), after processing A_2 , graph G_2 contains six nodes. Nodes d_0 , d_1 , d_2 , and d_4 are*

Fairness Layer Thread (Ordering transactions) :

```

1: function ORDERFINALIZATION() do
2:   while  $G_r := graphs.Front()$  do
3:     if  $G_r$  is a tournament then
4:        $graphs.Pop()$ 
5:        $G_r^c := Trajan\_SCC(G_r)$ 
6:        $[S_1, S_2, \dots, S_s] := Topologically\_Sorted(G_r^c)$ 

7:        $last := \max\{j \mid \exists node \in S_j, node.type = solid\}$ 
8:       for  $j = 1, 2, \dots, last$  do
9:          $p_j := Hamilton\_Path(S_j)$ 
10:        Append  $p_j$  to final ordering
11:        for  $node(d) \in p_j$  do
12:           $ordered\_nodes.add(node(d))$ 

13:       $G_{r'} := graphs.Front()$ 
14:      for  $j = last + 1, last + 2, \dots, s$  do
15:        for  $node(d) \in S_j$  do
16:           $ap(d, r') := |\{i \mid node(d).committed\_rounds[i] \leq r'\}|$ 
17:          if  $ap(d, r') \geq n - f$  then
18:             $node(d).type = solid$ 
19:          else if  $ap(d, r') \geq \frac{n-f}{2}$  then
20:             $node(d).type = shaded$ 
21:             $G_{r'}.nodes.add(node(d))$ 
22:            for  $node(d_2) \in G_{r'}.nodes$  do
23:              Calculate  $G_{r'}.weights[(d, d_2)]$ 
24:              Calculate  $G_{r'}.weights[(d_2, d)]$ 
25:              if  $G_{r'}.weights[(d, d_2)] \geq \frac{n-f}{2}$ 
26:                and  $\vee G_{r'}.weights[(d_2, d)] \geq \frac{n-f}{2}$  then
27:                   $G_{r'}.edges.add(e(d, d_2))$ 
28:                else
29:                   $G_{r'}.edges.add(e(d_2, d))$ 
30:            else
31:              break

```

FIGURE 5.11. Finalizing Transaction Ordering in FairDAG-RL.

classified as solid (solid circles), while d_3 and d_5 are shaded (dashed circles). All node pairs form edges except (d_3, d_5) , as neither $G_2.weight[(d_3, d_5)]$ nor $G_2.weight[(d_5, d_3)]$ reaches the threshold $\frac{n-f}{2} = \frac{3}{2}$. In Figure 5.9(b), after processing A_4 , G_4 is constructed with two additional nodes, and an edge between $node(d_3)$ and $node(d_5)$ is added once $G_2.weight[(d_3, d_5)]$ reaches the threshold.

5.5.3. Ordering Finalization. A *tournament* graph is a graph such that there is an edge between each pair of nodes. After adding edges, we check if the graphs are *tournaments* in a round-increasing order. If a graph is a *tournament*, we finalize the ordering of transactions within it as follows and then check the next graph until encountering a *non-tournament* graph.

Condensing dependency graph

In graph theory, a *strongly connected component (SCC)* is a maximal subset of nodes such that for every pair of nodes ($node_1, node_2$) in the subset, there exists a directed path from $node_1$ to $node_2$ and a directed path from $node_2$ to $node_1$.

For a *tournament* dependency graph G_r , we condense it into G_r^c by finding all *SCCs* with *Tarjan's SCC algorithm*. Figure 5.9 (c) shows G_2^c the condensed graph of G_2 .

Ordering finalization

It is guaranteed that after condensation, there will be one and only one topological sorting of the *SCCs* in G_r^c , which we denote by $S_1, S_2, \dots, S_s$, where S_j represents the j -th *SCC* in the topological sorting.

We denote by the S_{last} the last *SCC* that contains a *solid* node. Then, for each S_j such that $j \leq last$, we find a *Hamilton path* p_j of nodes in S_j and append p_j to the final transaction ordering. In Figure 5.9 (c), S_2 is S_{last} of G_2^c .

Readding shaded nodes

For the *SCCs* behind S_j , the nodes are all *shaded* nodes. We add these nodes into the next graph $G_{r'}$. For each $node(d)$, we run the following steps:

- (1) Classify $node(d)$ based on $ap(d, r')$;
- (2) Calculate the weights between $node(d)$ and all existing nodes in $G_{r'}$.
- (3) Add edges between $node(d)$ and other nodes if the weight reaches threshold $\frac{n-f}{2}$.

In Figure 5.9 (d), $node(d_5)$ is readded into G_4 , being classified as a *solid* node. And two edges pointing from $node(d_5)$ are added.

5.5.4. Ordering Dependency. After illustrating how to construct dependency graphs and finalize transaction ordering in FAIRDAG-RL, we now discuss *ordering dependencies* in FAIRDAG-RL. For any transaction T_1 and T_2 with digests d_1 and d_2 , either T_1 is dependent on T_2 or T_2 is dependent on T_1 . As the transactions are ordered based on the edges in the dependency graphs, intuitively, if an edge $e(d_1, d_2)$ exists, then T_2 is dependent on T_1 .

However, for two transactions that are in different replicas, deciding *ordering dependency* is more complicated. We denote by $weight[(d_2, d_1)]_{max}$ the maximal number of the local orderings in which T_2 is ordered before T_1 . Even though $node(d_2)$ is in an earlier dependency graph, it is

possible that $weight[(d_2, d_1)]_{max} < \frac{n-f}{2}$, which implies that edge $e(d_2, d_1)$ cannot exist even if the two nodes are in the same dependency graph. Then, if $weight[(d_2, d_1)]_{max} < \frac{n-f}{2}$, T_2 is dependent on T_1 .

To endorse the transactions that are added as nodes earlier, if T_1 is in a earlier dependency graph and $weight[(d_1, d_2)]_{max} \geq \frac{n-f}{2}$, which implies that edge $e(d_1, d_2)$ could exist, we also say that T_1 is also dependent on T_2 .

DEFINITION 5.5.2. *We say that T_2 with digest d_2 is dependent on T_1 with digest d_1 if:*

- $weight[(d_2, d_1)]_{max} < \frac{n-f}{2}$; **or**
- *edge $e(d_1, d_2)$ exists; or*
- $weight[(d_1, d_2)]_{max} \geq \frac{n-f}{2}$ and *node(d_1) is in a earlier dependency graph.*

5.6. Comparing Fairness Protocols

In this section, we demonstrate how FAIRDAG-AB and FAIRDAG-RL outperform POMPE [169] and THEMIS [99] in limiting the adversary’s manipulation of transaction ordering. We achieve this by comparing how these protocols perform under adversarial conditions, such as those caused by Byzantine replicas or an asynchronous network.

5.6.1. Pompe and Themis. POMPE and THEMIS run fairness protocols atop leader-based consensus protocols such as PBFT [39] and HotStuff [164], where a single leader is responsible for collecting local orderings from a quorum of $n-f$ replicas.

Beyond the difference in underlying consensus protocols, POMPE assigns non-overlapping intervals to consensus rounds, allowing only transactions whose *AOI* fall within the corresponding interval to be executed, increasing the overhead of recovery when the leader fails. Additionally, in POMPE, each client transaction is sent to a single replica instead of being broadcast, making the protocol more susceptible to transaction censorship.

5.6.2. Adversarial Manipulation. We now analyze how an adversary can manipulate transaction ordering in POMPE and THEMIS by selectively collecting local orderings. Additionally, we demonstrate how FAIRDAG-AB and FAIRDAG-RL mitigate these vulnerabilities, ensuring more resilient and fair transaction orderings.

Pompe vs FairDAG-AB

In POMPE, a Byzantine replica who is responsible for collecting ordering indicators can selectively choose $\mathbf{n}-\mathbf{f}$ local ordering indicators to calculate an assigned ordering indicator. This selective collection allows the adversary to manipulate the ordering of transactions whose ranges of correct ordering indicators overlap.

Furthermore, if at most $\mathbf{f}$ correct replicas have received the assigned ordering indicator for a transaction T before a Byzantine leader replica collects assigned ordering indicators for a new block, the leader can exclude T from the new block. Since the leader is only required to collect from $\mathbf{n}-\mathbf{f}$ replicas, it can selectively collect from $\mathbf{n}-\mathbf{f}$ replicas that have not received T , thus delaying the position of T in the final ordering.

In FAIRDAG-AB, the client broadcasts its transaction to all replicas, and each replica independently broadcasts its ordering indicators. The *Validity* property and the round-robin or random leader rotation of the underlying DAG protocols guarantees that reduce the chance of selective collection of ordering indicators by Byzantine replicas. Thus, FAIRDAG-AB is more resilient against ordering manipulation than POMPE.

Additionally, POMPE effectively mitigates the transaction censorship issue mentioned in Section 5.6.1, because transactions are broadcast to all replicas, and each replica independently generates and broadcasts its local ordering indicators.

Themis vs FairDAG-RL

In THEMIS, if a Byzantine leader ignore $\mathbf{f}$ correct local orderings containing transaction T and the Byzantine replicas exclude T from their local orderings, there would be at most $\mathbf{n}-2\mathbf{f}$ local orderings containing T . Thus, the threshold of deciding edge direction is $\frac{\mathbf{n}-2\mathbf{f}}{2}$ in THEMIS. To guarantee the γ -Batch-Order-Fairness, it is required that the votes of the opposite direction cannot reach the threshold, containing $\mathbf{f}$ votes from Byzantine replicas and $(1-\gamma)(\mathbf{n}-\mathbf{f})$ votes from correct replicas. That is, $\mathbf{f} + (1-\gamma)(\mathbf{n}-\mathbf{f}) < \frac{\mathbf{n}-2\mathbf{f}}{2}$, i.e., $\mathbf{n} > \frac{\mathbf{f}(2\gamma+2)}{2\gamma-1}$. When $\gamma = 1$, THEMIS requires $\mathbf{n} > 4\mathbf{f}$.

In FAIRDAG-RL, due to the Validity of DAG-based consensus protocols, all local orderings from correct replicas will eventually be committed by all replicas. Then, for each transaction, there are at least $\mathbf{n}-\mathbf{f}$ local orderings containing it. Thus, the threshold for deciding edge direction is $\frac{\mathbf{n}-\mathbf{f}}{2}$ in THEMIS. Then, to guarantee the γ -Batch-Order-Fairness, it is required that $\mathbf{f} + (1-\gamma)(\mathbf{n}-\mathbf{f}) < \frac{\mathbf{n}-\mathbf{f}}{2}$, i.e., $\mathbf{n} > \frac{\mathbf{f}(2\gamma+1)}{2\gamma-1}$. When $\gamma = 1$, FAIRDAG-RL requires $\mathbf{n} > 3\mathbf{f}$.

5.6.3. Leader Crash and Asynchronous Network. In POMPE and THEMIS, if the leader crashes, a recovery subprocess must be initiated to replace the leader with a new one, introducing an additional delay of $O(\Delta)$ before the protocol can resume normal operation. Moreover, in POMPE, each round corresponds to a distinct, non-overlapping time slot. If the designated leader crashes, transactions with assigned ordering indicators that fall within that time slot cannot be ordered or executed, resulting in a potential transaction loss or indefinite delays. In THEMIS, if the leader crashes, replicas have to resend their local orderings to the new leader, resulting in additional communication overhead.

If the network operates under asynchronous conditions where messages can experience indefinite delays, additional overhead will be introduced, similar to the overhead incurred during leader crashes.

FAIRDAG-AB and FAIRDAG-RL address the aforementioned issues through the multi-proposer design and the Reliable Broadcast inherent to DAG-based consensus protocols, which ensures that each correct local ordering eventually delivers and commits even in asynchronous settings.

5.7. Correctness Proof

In this section, we will prove the safety, liveness and fairness properties of FAIRDAG-AB and FAIRDAG-RL. Derived from [49, 97, 150], the DAG-Layer has the following properties:

- **Agreement:** if a correct replica commits A_r of leader vertex L_r , then every other correct replica eventually commits A_r .
- **Total Ascending Order:** if a correct replica commits A_r before $A_{r'}$, then $r < r'$ and no correct replica commits $A_{r'}$ before A_r .
- **Validity:** if a correct replica broadcasts a DAG vertex v , then eventually each correct replica will commit a leader vertex L_r such that $v \in A_r$.

Combining the **Agreement** and **Total Ascending Order**, we have the following lemma:

LEMMA 5.7.0.1. *If a correct replica R commits a series of leader vertices $L_list^R = L_{r_j}^R, L_{r_{j+1}}^R, \dots$ and every other correct replica R' will eventually commit $L_list^{R'} = L_{r_j}^{R'}, L_{r_{i+1}}^{R'}, \dots$, such that $\forall j, r_j < r_{j+1}; L_{r_j}^R = L_{r_j}^{R'}; \text{ and } C_{r_j}^R = C_{r_j}^{R'}$.*

5.7.1. Safety.

LEMMA 5.7.0.2. *In FAIRDAG-AB, if a correct replica R assigns transaction T with digest d assigned ordering indicators $d.AOI^R$, then every other correct replica R' will eventually assign T with $d.AOI^{R'}$ such that $d.AOI^R = d.AOI^{R'}$.*

PROOF. According to FAIRDAG-AB algorithm, we know that a correct replica deterministically calculates $d.AOI$ based on C_r , the causal history of the lowest-round leader vertex L_r such that after the replica commits L_r , there are at least $\mathbf{n} - \mathbf{f}$ committed_ois of d in C_r .

Then, we prove the lemma by contradiction. We assume that R calculates $d.AOI^R$ based on the causal history C_r^R of leader vertex L_r^R . If R' never assigns T with an *assigned ordering indicator*, then R' never commits C_r^R . If R' assigns T with an *assigned ordering indicator* different from T' , then R' must have committed some different $C_r^{R'}$. Both cases contradict 5.7.0.1. $\square$

LEMMA 5.7.0.3. *In FAIRDAG-AB, after A_r is committed and processed, for a transaction T with digest d that has no assigned ordering indicator but a lowest possible assigned ordering indicator $d.LPAOI_r$, if T eventually gets an assigned ordering indicator $d.AOI$, then $d.AOI \geq d.LPAOI_r$.*

PROOF. We denote by *highest_oi_list_r* and *d.seen_ois_r*, respectively, the *highest_oi_list* and *d.seen_ois* after A_r is committed. According to the FAIRDAG-AB algorithm, $d.LPAOI_r$ is the $(\mathbf{f} + 1)$ -th lowest value of *lp_ois_r* where $lp_ois_r[i] := \min(d.seen_ois_r[i], highest_oi_list_r[i])$, $1 \leq i \leq \mathbf{n}$. As the DAG grows, each new $d.committed[i]$ will be not smaller than $\min(d.seen_ois_r[i], highest_oi_list_r[i])$. Thus, as $d.AOI := sorted(d.committed_ois[i])(\mathbf{f} + 1)$, where $\forall i, d.committed_ois[i] \geq lp_ois_r[i]$, it holds true that $d.AOI \geq d.LPAOI_r$. $\square$

LEMMA 5.7.0.4. *In FAIRDAG-AB, for any two transactions T_1 and T_2 with digests d_1 and d_2 , if $d_1.AOI < d_2.AOI$, then d_1 will be ordered before d_2 in the final ordering.*

PROOF. We denote by or_j the round such that transaction digest d_j is ordered, getting an *assigned ordering indicator* lower than $LPAOI_{min}$, i.e., $d_j.AOI < LPAOI_{min}$.

Now we prove by the lemma by contradiction. Assuming that d_2 is ordered before d_1 , then obviously, $or_1 \geq or_2$.

- If $or_1 = or_2$, then according to FAIRDAG-AB algorithm, d_1 and d_2 are ordered based on assigned ordering indicator. Thus, with a lower AOI , d_1 would be ordered before d_2 , contradicting with our assumption.
- If $or_1 > or_2$, then by the definition of $LPAOI_{min}$ we know that $d_2.AOI < LPAOI_{min} \leq d_1.LPAOI_{or_2}$. Also, from

Lemma 5.7.0.3 we know that eventually d_1 will get $d_1.AOI \geq d_1.LPAOI_{or_2}$. Thus, we have $d_1.AOI > d_2.AOI$, which contradicts the fact that $d_1.AOI < d_2.AOI$.

In summary, it holds true that if $d_1.AOI < d_2.AOI$, then d_1 will be ordered before d_2 in the final ordering. $\square$

THEOREM 5.7.1. (FAIRDAG-AB SAFETY) *In FAIRDAG-AB, if a correct replica orders transaction T at position p in the final ordering, then every correct replica will eventually order T at position p .*

PROOF. From Lemma 5.7.0.2 we know that every correct replica will eventually assign the same *assigned ordering indicator* to T . From Lemma 5.7.0.4 we know that all transactions with *assigned ordering indicators* are ordered in an ascending order of AOI . Combining the two claims above, we conclude that every correct replica will eventually order T at the same position in the final ordering. $\square$

THEOREM 5.7.2. (FAIRDAG-RL SAFETY) *In FAIRDAG-RL, if a correct replica orders transaction T at position p in the final ordering, then every correct replica will eventually order T at position p .*

PROOF. After committing a leader vertex, each correct replica uses a deterministic method to construct dependency graphs and finalize transaction order. Combining this with Lemma 5.7.0.1, we know that safety holds for FAIRDAG-RL. $\square$

5.7.2. Liveness. We claim the following assumption holds, which is necessary for the liveness property.

ASSUMPTION. If a correct replica R receives transaction T , then every correct replica will eventually receive transaction T .

LEMMA 5.7.2.1. *In FAIRDAG-AB, if a transaction T with transaction d is received by correct replicas, then T will eventually get an assigned ordering indicator.*

From Assumption 5.7.2 we know that all correct replicas will receive T will propose a DAG vertex containing d and a corresponding ordering indicator. From the **Validity** of DAG layer, we know that all these DAG vertices will be committed. Thus, d will get at least $\mathbf{n} - \mathbf{f}$ *committed_ois* from correct replicas and then get an *assigned ordering indicator*.

LEMMA 5.7.2.2. *For transactions T_1 and T_2 with digests d_1 and d_2 , if only T_1 has an assigned ordering indicator and T_2 has an $LPAOI$ lower than $d_1.AOI$, i.e., $d_2.AOI = \infty \wedge d_2.LPAOI < d_1$, then eventually,*

- *either T_2 gets an assigned ordering indicator;*
- *or $d_2.LPAOI$ becomes larger than $d_1.AOI$.*

PROOF. If T_2 is received by correct replicas, from Lemma 5.7.2.1, we know that eventually T_2 gets an *assigned ordering indicator*.

Next, we discuss the case that T_2 is received by only faulty replicas. Since $d_2.LPAOI$ is the $(\mathbf{f} + 1)$ -th lowest value of *lp_ois*, $\mathbf{n} - \mathbf{f}$ values of which are the *highest_oi_list* values from correct replicas, thus, $d_2.LPAOI$ is not greater than *highest_oi_list* value from at least one correct replica. Due to the **Valifity** property of the DAG layer, DAG keeps growing and the *highest_oi_list* value from each correct replica will eventually be higher than $d_1.AOI$. $\square$

THEOREM 5.7.3. (FAIRDAG-AB Liveness) *If a transaction T with digest d is received by correct replicas, then T will eventually be ordered.*

PROOF. From Lemma 5.7.2.1 we know that T will eventually get an $d.AOI$. From Lemma 5.7.2.2 we know that eventually there will be no other transaction that has an $LPAOI$ lower than $d.AOI$. Thus, eventually it will be satisfied that $d_1.AOI < LPAOI_{min}$, and then T will be ordered. $\square$

LEMMA 5.7.3.1. *In FAIRDAG-RL, each dependency graph G will eventually become a tournament.*

PROOF. Since only *solid* and *shaded* nodes can be added into a dependency graph, for each $node(d)$ in G , there are at least $\frac{n-f}{2} > f + 1$ local orderings that contain d . Thus, d will be received by all correct replicas. And then eventually, due to the **Validity** property of the DAG layer, there will be a round r such that $ap(d, r) \geq n - f$.

Thus, for each pair of nodes $node(d_1)$ and $node(d_2)$ in G , there will be a round r such that $ap(d_1, r) \geq n - f$ and $ap(d_2, r) \geq n - f$. Thus, at least one of $G.weights[(d_1, d_2)]$ and $G.weights[(d_2, d_1)]$ will reach the threshold $\frac{n-f}{2}$. Then, eventually, there will be an edge between each pair of nodes in G , i.e., G will be a tournament. $\square$

THEOREM 5.7.4. (FAIRDAG-RL Liveness) *If a transaction T with digest d is received by correct replicas, then T will eventually be ordered.*

PROOF. Due to the **Validity** property of the DAG layer, there will eventually be a round r such that $ap(d, r) \geq n - f$, and then $node(d)$ will be added into a dependency graph G .

From Lemma 5.7.3.1 we know that G will eventually be a tournament. If $node(d)$ is *solid*, the T will be ordered. If $node(d)$ is *shaded*, the T might be ordered. Even if *shaded* $node(d)$ has to be added into the next dependency graph, eventually $node(d)$ will be added as a *solid* node and T will be ordered. $\square$

5.7.3. FairDAB-AB: Ordering Linearizability. To prove ordering linearizability, we introduce the following denotations:

- $d._ois^C$: the ordering indicators of d from correct replicas.
- $d.low_oi^C$: the lowest value in $d._ois^C$.
- $d.high_oi^C$: the highest value in $d._ois^C$.

LEMMA 5.7.4.1. *For each transaction T with digest d , $d.low_oi^C \leq d.AOI \leq d.high_oi^C$.*

PROOF. As the $d.AOI$ is the $(f + 1)$ -th lowest value of a subset of $d.committed_ois$ with at least $n - f \geq 2f + 1$ values. Among the $f + 1$ lowest values of the subset, at least one is in $d._ois^C$. Thus, $d.low_oi^C \leq d.AOI$. Similarly, among the $f + 1$ highest values of the subset, at least one is from a correct replica. Thus, $d.AOI \leq d.high_oi^C$. $\square$

THEOREM 5.7.5. (Ordering Linearizability) *For two transactions T_1 and T_2 with digests d_1 and d_2 , if $d_1.high_oi^C < d_2.low_oi^C$, then T_1 will be ordered before T_2 in the final ordering.*

PROOF. From Lemma 5.7.4.1 we know that $d_1.AOI \leq d_1.high_oi^C$ and $d_2.low_oi^C \leq d_2.AOI$. Thus, $d_1.AOI < d_2.AOI$. From Lemma 5.7.0.4 we know that transactions are ordered based on AOI values. Thus, T_1 will be ordered before T_2 in the final ordering. $\square$

5.7.4. FairDAG-RL: γ -Batch-Order-Fairness.

LEMMA 5.7.5.1. *For any two transactions T_1 and T_2 with digests d_1 and d_2 , if $\gamma(\mathbf{n} - \mathbf{f})$ correct replicas receive T_1 before T_2 , then $\frac{\mathbf{n} - \mathbf{f}}{2} > weight[(d_2, d_1)]_{max}$.*

PROOF. As $\gamma(\mathbf{n} - \mathbf{f})$ correct replicas receive T_1 before T_2 , then $weight[(d_2, d_1)]_{max} \geq \mathbf{f} + (1 - \gamma)(\mathbf{n} - \mathbf{f})$. As $\mathbf{n} > \frac{(2\gamma+1)\mathbf{f}}{(2\gamma-1)}$, $\frac{1}{2} < \gamma \leq 1$, we have:

$$\begin{aligned} \mathbf{n} > \frac{(2\gamma+1)\mathbf{f}}{(2\gamma-1)} &\iff (2\gamma-1)(\mathbf{n} - \mathbf{f}) > 2\mathbf{f} \iff (2\gamma-2)(\mathbf{n} - \mathbf{f}) + \mathbf{n} - \mathbf{f} > 2\mathbf{f} \iff \mathbf{n} - \mathbf{f} > \\ 2\mathbf{f} + (2-2\gamma)(\mathbf{n} - \mathbf{f}) &\iff \frac{\mathbf{n} - \mathbf{f}}{2} > \mathbf{f} + (1-\gamma)(\mathbf{n} - \mathbf{f}) \iff \frac{\mathbf{n} - \mathbf{f}}{2} > weight[(d_2, d_1)]_{max} \end{aligned}$$

$\square$

Combing Lemma 5.7.5.1 with the definition of *ordering dependency*, we have:

LEMMA 5.7.5.2. *For any two transactions T_1 and T_2 with digests d_1 and d_2 , if $\gamma(\mathbf{n} - \mathbf{f})$ correct replicas receive T_1 before T_2 , then T_2 is dependent on T_1 , i.e., $T_1 \rightarrow T_2$.*

LEMMA 5.7.5.3. *For any two transactions T_1 and T_2 with digests d_1 and d_2 in a tournament dependency graph G , if $\gamma(\mathbf{n} - \mathbf{f})$ correct replicas receive T_1 before T_2 , then after condensing G and topologically sorting the SCCs, $node(d_1)$ is in the same or an earlier SCC than $node(d_2)$.*

PROOF. According to graph theory, we know that a tournament dependency graph G can be condensed into a graph with multiple *SCCs*, and after topologically sorting the *SCCs*, there is a unique list $S_1, S_2, \dots, S_s$. It holds that for any two *SCCs* S_a and S_b , if $a < b$, then $\forall node(d_a) \in S_a, \forall node(d_b) \in S_b$, edge $e(d_a, d_b)$ exists in G .

We prove the lemma by contradiction. Assuming that $node(d_2)$ is in an earlier *SCC* than $node(d_1)$, then $e(d_2, d_1)$ exists in G . However, it contradicts Lemma 5.7.5.1, from which we know $\frac{\mathbf{n} - \mathbf{f}}{2} > weight[(d_2, d_1)]_{max}$, i.e., $e(d_2, d_1)$ cannot exist. $\square$

LEMMA 5.7.5.4. $\forall node(d_1) \in G_{r_1}, \forall node(d_1) \in G_{r_2}, r_1 < r_2$, if $node(d_1)$ is solid, then transaction T_2 is dependent on T_1 .

PROOF. If $node(d_2)$ was not added as a node into G_{r_1} , then $ap(d_2, r_1) < \frac{n-f}{2}$ when $ap(d_1, r_1) \geq n - f$. Thus, there are more than $\frac{n-f}{2}$ committed local orderings in which T_1 is before T_2 . Then $weight[(d_1, d_2)]_{max} \geq \frac{n-f}{2}$, then, combining with the fact that $node(d_1)$ is in an earlier dependency graph, T_2 is dependent on T_1 .

If $node(d_2)$ was added as a node into G_{r_1} but readded into G_{r_1} , then it implies that $node(d_2)$ was in an *SCC* later than the last *SCC* that contains a *solid* node. Thus, *solid* $node(d_1)$ has an edge to $node(d_2)$, i.e., $e(d_1, d_2)$ exists, and then T_2 is dependent on T_1 . $\square$

THEOREM 5.7.6. (*γ -Batch-Order-Fairness*) For any two transactions T_1 and T_2 with digests d_1 and d_2 , if $\gamma(n - f)$ correct replicas receive T_1 before T_2 , then T_1 will be ordered no later than T_2 .

PROOF. We denote by G_{r_1} and G_{r_2} in which T_1 and T_2 are ordered, respectively. If $r_1 < r_2$, then obviously T_1 is ordered before T_2 in the final transaction ordering. If $r_1 = r_2$, then from Lemma 5.7.5.4 we know that $node(d_1)$ is in the same or an earlier *SCC* than $node(d_2)$. Thus, T_1 is ordered no later than T_2 in the final transaction ordering.

If $r_1 > r_2$, then $node(d_2)$ is *shaded*. Otherwise, assuming $node(d_2)$ is *solid*, then there are at least $n - f$ committed local ordering containing d_2 when $node(d_2)$ is added into G_{r_2} . From Lemma 5.7.5.1 we know that $G_{r_2}.weights[(d_2, d_1)] < \frac{n-f}{2}$, then $G_{r_2}.weights[(d_1, d_2)] > \frac{n-f}{2}$, and edge $e(d_1, d_2)$ exists. Therefore, $node(d_1)$ should be ordered in G_{r_2} as it has a path to a *solid* node, contradicting the fact that $r_1 > r_2$. Thus, $node(d_2)$ must be *shaded* in G_{r_2} . Hence, there is some *solid* node $node(d_s)$ such that there is a path p_2 from $node(d_2)$ to $node(d_s)$. We denote by p_1 the path in G_{r_1} from the first ordered node to $node(d_1)$. Combining the following information:

- from Lemma 5.7.5.4 we know that all transactions on p_1 , including $T - 1$, are dependent on transaction T_s of $node(d_s)$;
- from Lemma 5.7.5.2 we know that T_2 is dependent on T_1 .
- along path p_2 , each transaction is dependent on the previous one, from T_s to T_2 .

Thus, the transactions on p_1 and p_2 form a *cyclic dependent batch* b . For the transactions that are ordered in G_{r_2} later than T_s , they are in the same *SCC* as T_s and then can be added into b . Therefore, even if $r_1 > r_2$, T_1 and T_2 are in the same *cyclic dependent batch* in the final ordering, i.e., T_1 is ordered *no later* than T_2 . $\square$

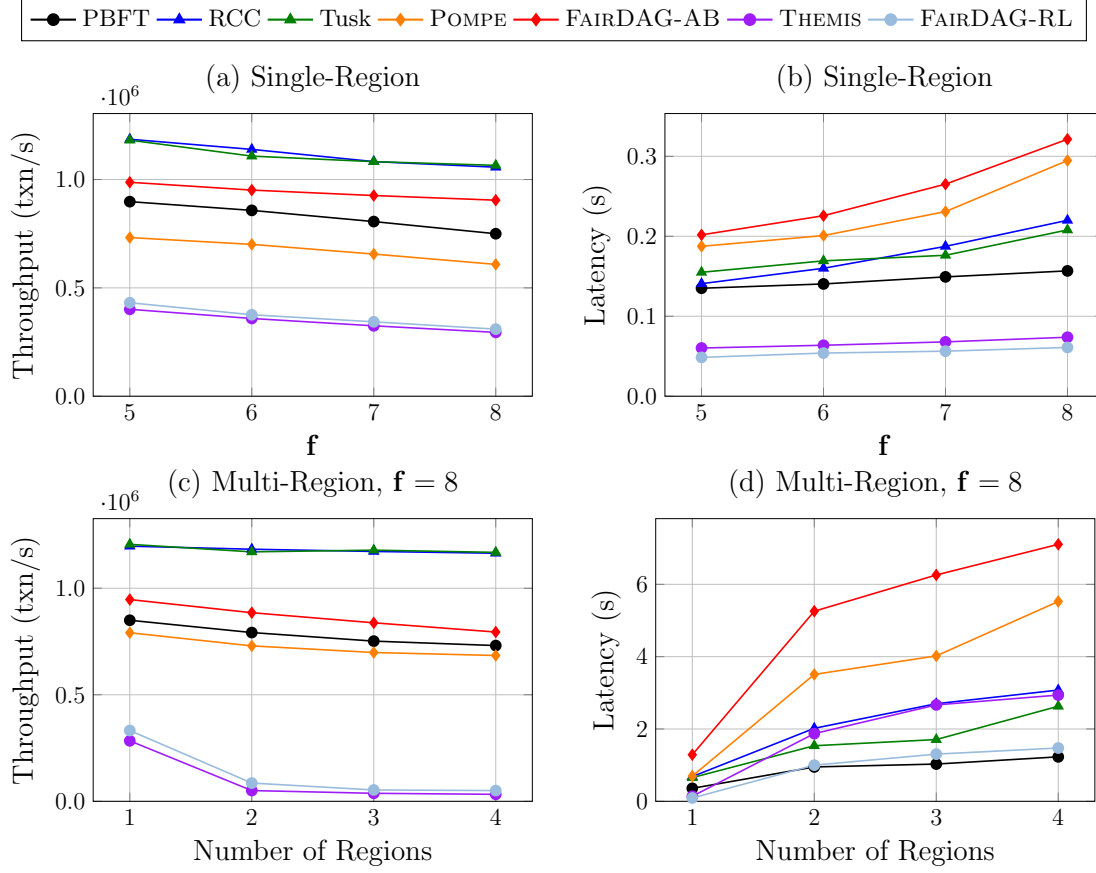


FIGURE 5.13. Performance of FAIRDAG and baseline protocols, with varying f (a,b), and varying number of regions (c,d).

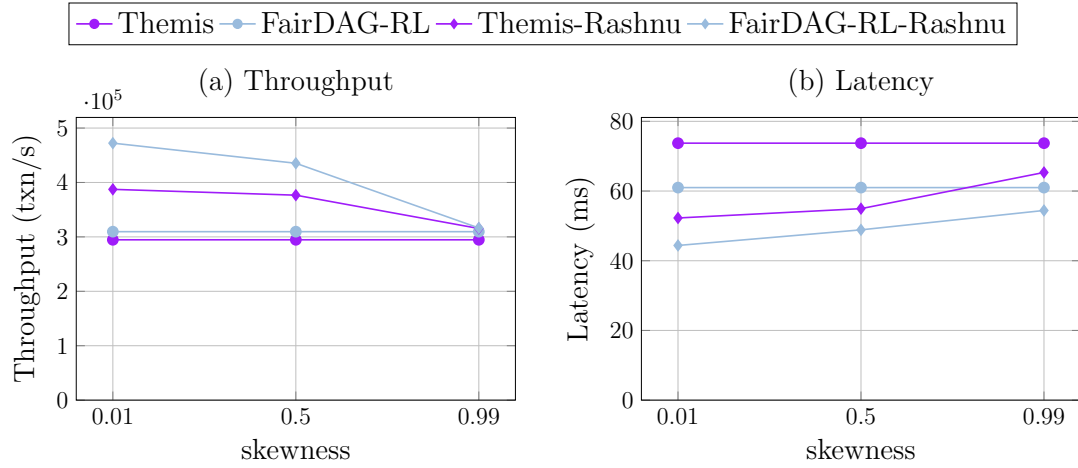


FIGURE 5.14. Performance of Rashnu-enhanced variants vs. relative fairness protocols.

- *Client Latency* – the average duration between the time a client sends a transaction and the time the client receives $f+1$ matching responses.

We compare the performance of the protocols with varying f , the maximum number of faulty replicas allowed, from 5 to 8. With the same f , different protocols have different minimum replica number requirements. For example, when $f = 5$, THEMIS requires $n = 21$ replicas, while other protocols require $n = 16$ replicas.

Besides design, the performance of the protocols is highly related to the workloads. As shown in Figure 5.12, where we set $f = 8$, as the workload increases, the throughput increases until the pipeline is fulfilled by the transactions. Then, after reaching the throughput limit, the latency increases as the workload increases. We define by *optimal point* the point with the lowest latency while maintaining the highest throughput. And we evaluate their scalability at the *optimal points* of the protocols with varying f .

Throughput. Figure 5.13 shows that TUSK and RCC achieve higher throughput than other protocols because they have multiple proposers and no overhead for fairness guarantees. Due to the fairness overhead, when $f = 5$ and $f = 8$, FAIRDAG-AB reaches 83.5% and 84.9% throughput of TUSK, while FAIRDAG-RL reaches 11.9% and 12.6% throughput of TUSK.

However, compared to POMPE and THEMIS, the multi-proposer design of the DAG layer brings FAIRDAG-AB and FAIRDAG-RL advantages in throughput. When $f = 5$ and $f = 8$, FAIRDAG-AB obtains 30.2% and 52.6% higher throughput than POMPE, respectively. Similarly, FAIRDAG-RL reaches 7.5% and 5.1% higher throughput than THEMIS.

Latency. Without the fairness overhead, TUSK and PBFT, as the underlying consensus protocols, have lower latency than the fairness protocols running on top of them.

With $f = 5$ and $f = 8$, FAIRDAG-AB latency is 7.1% and 8.3% higher than POMPE, because TUSK, the underlying DAG consensus protocol of FAIRDAG-AB, has a higher commit latency than PBFT, the underlying consensus protocol of POMPE.

FAIRDAG-RL has a latency close to THEMIS when $f = 5$. As f grows, FAIRDAG-RL has a lower latency than THEMIS, which is 20.9% lower when $f = 8$. FAIRDAG-RL achieves a lower latency because THEMIS needs f more correct replicas to guarantee fairness, which causes higher overhead for both consensus and ordering. By comparing the latency of THEMIS with $f = 6$

and FAIRDAG-RL with $\mathbf{f} = 8$, we can verify this claim: with the same replica number $\mathbf{n} = 25$, FAIRDAG-RL achieves a 4.6% higher latency than THEMIS.

Geo-distributed performance. We conducted experiments under geo-distributed settings by deploying the systems across multiple AWS regions. Specifically, we varied the number of regions from 1 to 4. The regions include North Virginia, Oregon, London, and Zurich. We fixed $\mathbf{f} = 8$ and deploys $\frac{\mathbf{n}}{k}$ replicas in each region, where k is the number of regions. Figure 5.13 (c, d) show that in the geo-distributed setting, the latencies of all the protocols are high and increase with the number of regions, caused by the high inter-regional message delays. Furthermore, FAIRDAG-AB has higher throughput than the other fairness protocols.

We found that for all protocols except the relative fairness protocols, increasing the batch size allowed us to achieve throughput values comparable to those in the single-region setting. However, in FAIRDAG-RL and THEMIS, a larger batch size leads to a higher overhead of the fairness layer, which increases quadratically with batch size. Moreover, while FAIRDAG-RL achieves only a 5.1% throughput improvement over THEMIS in a single-region setting, we observe that in the geo-distributed setting, FAIRDAG-RL outperforms THEMIS by at least 42.1%. This significant gain is attributed to the robust performance of the underlying multi-proposer DAG-based consensus protocol in geo-scale settings with limited bandwidth and higher message delays.

Data-dependent fairness. RASHNU [125] proposes a technique to reduce the overhead of the fairness layer in relative fairness protocols by computing edge directions between only data-dependent transactions. This method is orthogonal to both FAIRDAG-RL and THEMIS. We implemented two RASHNU-enhanced variants, called THEMIS-RASHNU and FAIRDAG-RL-RASHNU, and compared them to THEMIS and FAIRDAG-RL.

In this experiment, we implemented a transaction workload with keys following a Zipfian distribution. We fixed $\mathbf{f} = 8$ and varied the skewness parameter s from 0.01 to 0.99, where a higher skewness indicates greater data dependency between transactions. The results presented in Figure 5.14 show that the RASHNU variants outperform the non-RASHNU variants and perform better as the skewness decreases. This improvement stems from the reduced overhead in calculating edge directions when transactions are less interdependent.

5.8.2. Tolerance to Byzantine Behavior. Next, we will discuss the impact of Byzantine behaviors on the performance and transaction ordering fairness of the protocols.

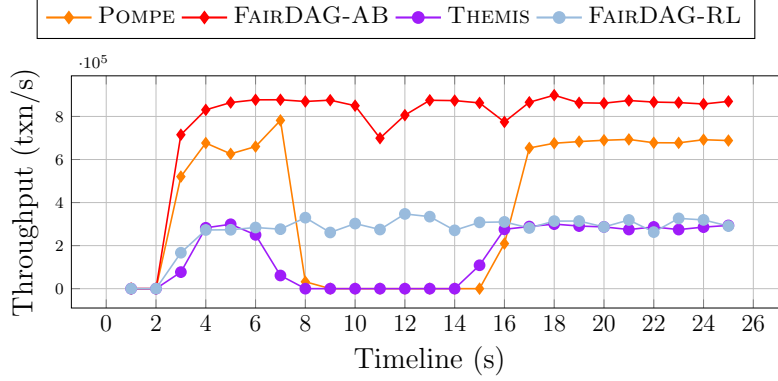


FIGURE 5.15. Real-Time throughput with a faulty leader.

Faulty leader. In this experiment, we make the consensus leader replica in POMPE and THEMIS faulty, which would trigger a view-change for leader replacement. While for FAIRDAG-AB and FAIRDAG-RL, we make a replica faulty due to the multi-proposer design. Figure 5.15 shows how the faulty leader affects the performance of THEMIS and POMPE. At time 7, the faulty leader stops sending any messages. After a period without progress, a view-change is triggered to replace the faulty leader. At time 15, the view-change is complete, and the throughput of POMPE and THEMIS recovers to the original level. In contrast, FAIRDAG-RL and FAIRDAG-AB are not affected because of the resilience provided by the multi-proposer design.

Adversarial Manipulation. We conduct two experiments in which Byzantine replicas attempt to manipulate transaction ordering. We evaluated the fairness quality of the fairness protocols under two Byzantine behaviors: (1) *Reversing order*: in each round, the Byzantine replicas reverse the local orderings of the transactions it has received. (2) *Targeted delay*: the Byzantine replicas intentionally delay targeted transactions by giving them higher local ordering indicators. In both cases, the Byzantine leader in POMPE and THEMIS excludes local orderings from $\mathbf{f}$ correct replicas.

For two transactions T_1 and T_2 , we say that they are correctly ordered if T_1 is ordered before T_2 and:

- (1) in relative fairness protocols, $weight[(d_1, d_2)] > \frac{n}{2}$;
- (2) in absolute fairness protocols, the $\mathbf{f}+1$ -th smallest local ordering indicator of d_1 is not larger than that of d_2 .

For the *reversing order* attack, we consider all transaction pairs; for the *targeted delay* attack, we consider only the transaction pairs that involve the targeted transactions.

For FAIRDAG-RL and THEMIS, we measure the ratio of correctly ordered pairs with different *Dist* values, where $Dist(d_1, d_2) = |weight[(d_1, d_2)] - weight[(d_2, d_1)]|$.

For FAIRDAG-AB and POMPE, we measure the ratio of correctly ordered pairs with different *Diff* values, where $Diff(d_1, d_2)$ is the minimal number of Byzantine local ordering indicators needed to order d_1 before d_2 . Formally, we denote by $asc_ois_1^C[i]$ the i -th *smallest* value in the correct local ordering indicators of d_1 , and by $desc_ois_2^C[j]$ the j -th *largest* value in the correct local ordering indicators of d_2 . To order d_1 before d_2 , we need to find (1) a pair of i and j such that $asc_ois_1^C[i] < desc_ois_2^C[j]$, (2) $\mathbf{f}+1-i$ Byzantine local ordering indicators smaller than $asc_ois_1^C[i]$, and (3) $\mathbf{f}+1-j$ Byzantine local ordering indicators larger than $desc_ois_2^C[j]$. Thus, we have $Diff(d_1, d_2) = \min\{2(\mathbf{f}+1)-i-j \mid asc_ois_1^C[i] < desc_ois_2^C[j]\}$, which is the minimal number of Byzantine local ordering indicators needed. We only consider transaction pairs with $Diff(d_1, d_2) > 0$.

We set $\mathbf{f} = 10$ and vary $\mathbf{f}_a$, the actual number of Byzantine replicas, from 0 to 10. For example, THEMIS-7 denotes THEMIS with $\mathbf{f}_a = 7$. As shown in Figure 5.16, FAIRDAG-RL and FAIRDAG-AB consistently demonstrate better resilience against adversarial ordering manipulation in all experimental settings, compared to THEMIS and POMPE, respectively. The results substantiate our claim that FAIRDAG effectively mitigates adversarial ordering manipulation through the properties inherent in the DAG-based consensus layer.

5.9. Related Work

Fairness in BFT

In traditional Byzantine Fault Tolerance (BFT) research, protocols [14, 39, 74, 83] are designed to ensure both safety and liveness in the presence of malicious replicas. Although these protocols do not explicitly guarantee fair transaction ordering, they mitigate unfair ordering to some extent. Protocols such as HotStuff [164], which employ leader rotation in a round-robin manner [32, 60, 65, 78, 93, 94, 118], provide each replica with the opportunity to propose a block.

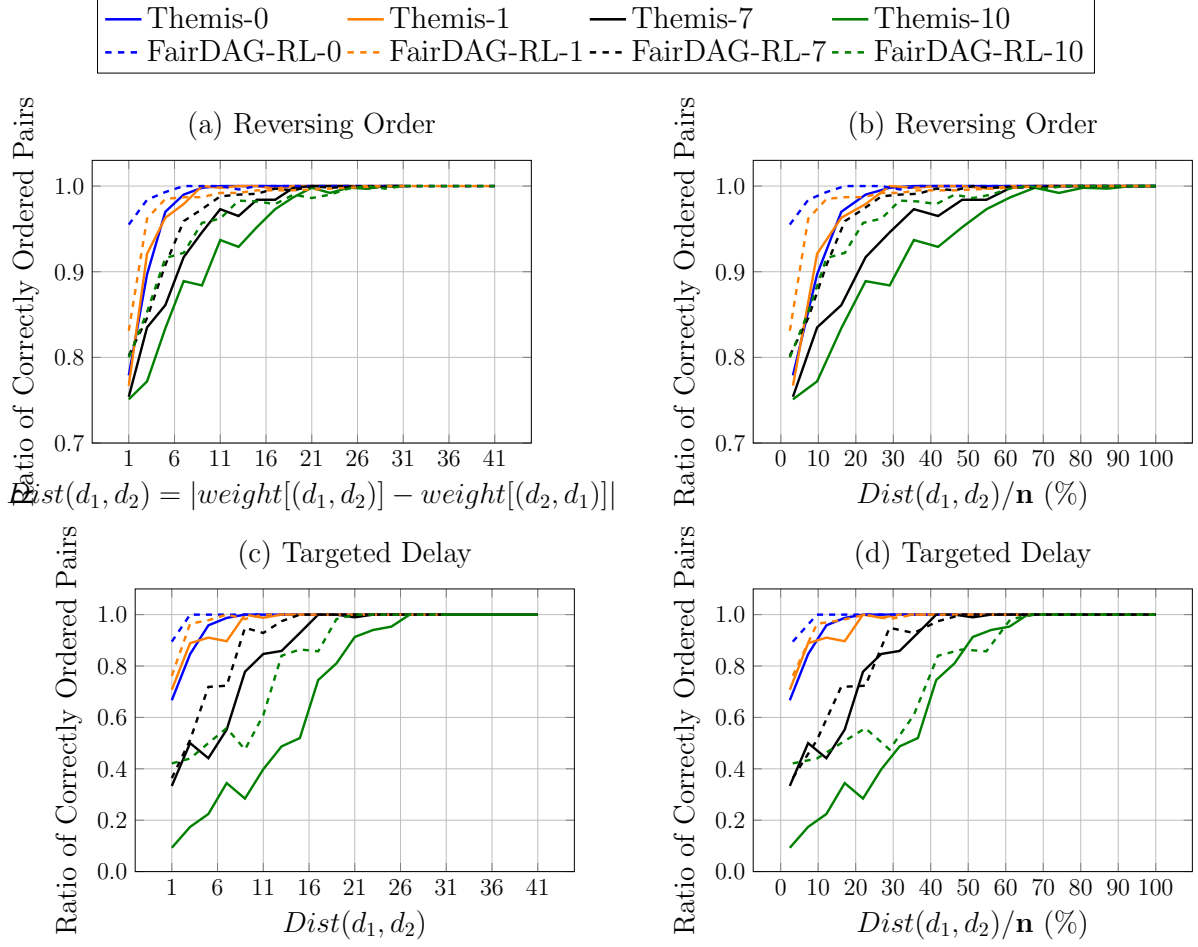


FIGURE 5.16. Fairness quality of relative fairness protocols under adversarial ordering manipulation.

Multi-proposer approaches, including concurrent consensus protocols [64, 77, 94, 151] and DAG-based protocols [18, 23, 49, 97, 149, 150, 162], enable multiple replicas to propose blocks concurrently, ordering them globally through predetermined or randomized mechanisms. Although these protocols reduce the reliance on a single leader and distribute transaction ordering authority, a Byzantine replica can still manipulate the ordering of transactions within the blocks it proposes.

Some protocols seek to eliminate the block proposers' oligarchy over the ordering of transactions within blocks via *censorship resistance* [10, 19, 34, 59, 111, 120, 123, 153, 162, 166]. In these protocols, a transaction is encrypted until the ordering of the transaction is determined. However, block proposers can still engage in censorship based on metadata, such as IP addresses, or prioritize their own transactions, knowing the content of encrypted transactions.

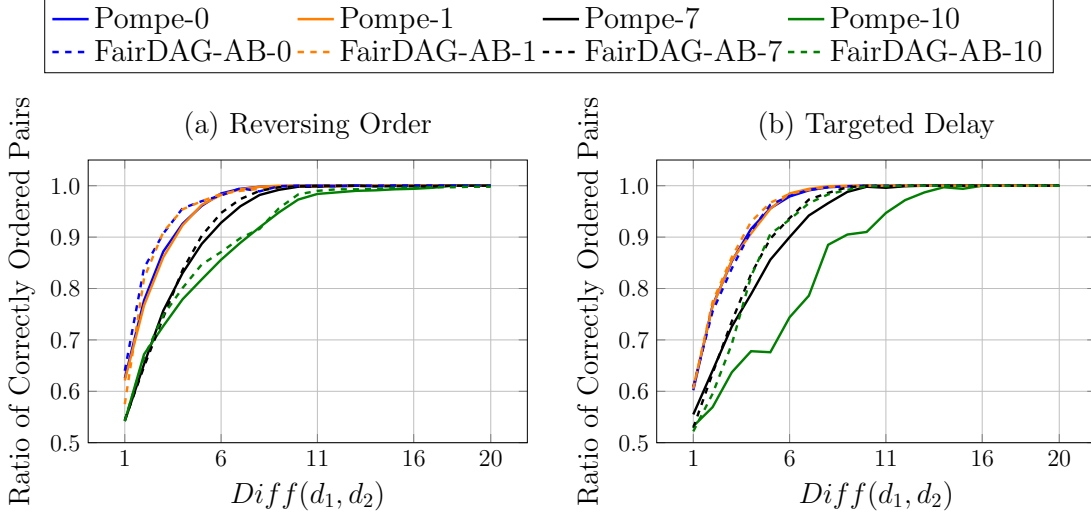


FIGURE 5.17. Fairness quality of absolute fairness protocols under adversarial ordering manipulation.

There are several prior fairness protocols that generate final ordering with collected local orderings. Wendy [104] guarantees *Timed-Relative-Fairness* similar to *Ordering Linearizability*, but it relies on synchronized local clocks, which are impractical in asynchronous networks. DCN [45] reaches δ -Median Fairness such that T_1 can be ordered before T_2 if T_1 is sent long enough earlier than T_2 . Aequitas [100] guarantees batch-order-fairness but suffers from liveness issues due to the existence of infinite *Condorcet Cycles*, which THEMIS solves via a *batch unspooling* mechanism. Quick-Order-Fairness [35] reaches batch-order-fairness with $\mathbf{n} > 3\mathbf{f}$ replicas but incurs $O(\mathbf{n}^3)$ communication complexity for the consensus leader. Rashnu [125] improves THEMIS performance by guaranteeing γ -Batch-Order-Fairness between only data-dependent transactions, but suffers from the same problems as THEMIS. SpeedyFair [124] pipelines the consensus layer and fairness layer, but still relies on a single leader to collect local orderings. *Ambush* attacks are identified in [133], which FAIRDAG-RL inherently mitigates with the underlying DAG-based consensus.

Multi-Proposer protocols

A significant amount of research [18, 23, 147, 149] has been dedicated to reducing the latency in DAG-based protocols. These works employ various techniques, including *pipelining DAG waves*, *fast commit rules*, *multi-anchor*, and *uncertified DAG*, to enhance the efficiency and speed of these protocols.

Concurrent consensus protocols [77, 94, 151] represent a distinct category of multi-proposer consensus protocols. These protocols run multiple concurrent consensus instances independently, generating a final ordering by globally ordering the committed blocks in each instance, round by round. However, these protocols face several challenges that make them unsuitable as the underlying consensus mechanisms for fairness protocols. First, the presence of a straggler instance, which lags behind the others, can substantially decrease throughput and increase system latency. Second, since the progress of different instances is not synchronized, a malicious leader of an instance could manipulate transaction ordering by delaying the block proposal until other replicas have proposed their blocks. DAG-based protocols address these issues effectively, as a replica can proceed to the next round once there are $n - f$ committed DAG vertices in the current round, ensuring more efficient and reliable progression.

5.10. Conclusion

In this chapter, we introduced FAIRDAG, a fair-ordering framework designed to run fairness protocols atop DAG-based consensus protocols. Through theoretical demonstration and experimental evaluation, we show that unlike previous fairness protocols, FAIRDAG-AB and FAIRDAG-RL, the two variants of FAIRDAG, not only uphold fairness guarantees, but also achieve better performance under normal and adversarial conditions, effectively constraining adversarial manipulation of transaction ordering.

CHAPTER 6

FEAT: Fair and Efficient Adversarial Transaction Ordering

Chapter 5 addressed the *architectural* limitation of leader-based fair-ordering protocols by replacing the leader-based consensus layer with a multi-proposer DAG-based one. A deeper *algorithmic* question remains, however, and is independent of the consensus layer: *what is the inherent complexity of enforcing γ -Batch-Order-Fairness?* The material in this chapter appeared as [52].

For reference, we recall the property:

DEFINITION 6.0.1. γ -Batch-Order-Fairness: *For any two transactions T_i and T_j in $\mathcal{T}$, if at least a fraction γ of correct replicas receive T_i before T_j , then T_i must be ordered no later than T_j in the final transaction ordering of $\mathcal{T}$.*

We observe that all existing fair-ordering protocols that guarantee γ -Batch-Order-Fairness [35, 92, 99, 100] share a common step: each constructs a complete pairwise precedence relation among transactions. Maintaining and processing such a complete relation typically incurs quadratic complexity in the size $|\mathcal{T}|$ of the transaction set. This raises the following fundamental question: *Is this quadratic cost inherent to enforcing γ -Batch-Order-Fairness, or merely an artifact of current protocol designs?*

In this chapter, we study the complexity of enforcing γ -Batch-Order-Fairness. We represent fairness constraints as a global pairwise comparator over transactions. We make the following contributions:

- (1) We define γ -Induced Comparator and Adjacent-Comparator Consistency, where Adjacent-Comparator Consistency is a sufficient condition for γ -Batch-Order-Fairness with a γ -Induced Comparator.
- (2) We prove that enforcing Adjacent-Comparator Consistency incurs a comparison complexity of $\Omega(|\mathcal{T}| \log |\mathcal{T}|)$ in the worst case over comparators.

- (3) We present FEAT—a fair-ordering algorithm that implements a γ -*Induced Comparator*, guarantees *Adjacent-Comparator Consistency*, and achieves $O(|\mathcal{T}| \log |\mathcal{T}|)$ per-round comparison complexity via an AVL-tree-based construction [9], thereby proving tightness.

The remainder of this chapter is organized as follows. Section 6.1 discusses background and related work. Section 6.2 presents the system model. Section 6.3 formalizes the fairness problem. Section 6.4 describes the design of FEAT. Section 6.5 establishes correctness and tight comparison complexity bounds of FEAT. Finally, Section 6.6 concludes.

6.1. Background

Existing fair-ordering protocols that guarantee the fairness property γ -*Batch-Order-Fairness* or its stronger variants, including AEQUITAS [100], THEMIS [99], QUICK-ORDER-FAIRNESS [35], and FAIRDAG-RL [92], calculate a complete ordering relation among transactions, which can be regarded as a tournament-shaped dependency graph (i.e., one in which any pair of vertices is connected by exactly one directed edge). This dependency graph is then traversed to produce a final linear ordering of transactions consistent with these dependencies.

Constructing such a tournament dependency graph incurs a quadratic cost in the number of transactions, both in computation and storage. In addition, since the dependency graph is constructed on top of a tournament structure with cycles allowed, the graph traversal often requires expensive operations such as cycle detection, yielding the final transaction ordering, e.g., finding a Hamilton path.

Beyond the protocols discussed above, some prior works pursue fair transaction ordering with alternative approaches. WENDY [104], POMPE [169], and FAIRDAG-AB [92] provide a weaker fairness guarantee, namely *Ordering Linearizability*. This property states that the final transaction ordering must respect the order of any two transactions whose order can be agreed upon by the majority of replicas. RASHNU [125] reduces the comparison complexity of THEMIS by enforcing γ -*Batch-Order-Fairness* only among data-dependent transactions, but incurs a quadratic complexity to detect data dependencies between transactions.

Some protocols seek to eliminate the block proposers’ oligarchy over the ordering of transactions within blocks via *censorship resistance* [10, 19, 34, 59, 111, 120, 123, 138, 153, 162, 166]. In these protocols, a transaction is encrypted until the ordering of the transaction is determined. However,

block proposers can still engage in censorship based on metadata, such as IP addresses, or prioritize their own transactions whose content they know.

A complementary line of work mitigates reordering attacks via random permutation: EATING SANDWICHES [11] applies a commit-reveal random permutation with transaction chunking, and BLINDPERM [96] composes an encrypted mempool with a randomized post-commit permutation.

Other approaches operate at the consensus layer: HOTSTUFF-1 [93] uses a slotting mechanism to narrow the proposer’s MEV window, CARRY-THE-TAIL [79] mitigates tail-forking MEV by carrying pending transactions across view changes, and multi-proposer protocols [23, 49, 64, 77, 94] distribute ordering control across concurrent proposers so that no single Byzantine proposer can unilaterally manipulate the order.

6.2. System Model

6.2.1. Replicas and Fault Model. We consider a distributed system consisting of a fixed set of $\mathbf{n}$ replicas. Up to $\mathbf{f}$ replicas may be *Byzantine/Adversarial*, exhibiting arbitrary malicious behavior, while the remaining replicas are *correct* and follow the protocol specification.

Transactions are disseminated through authenticated broadcast. Each replica locally observes a sequence of transactions, which may differ across replicas due to network delays or adversarial interference. Correct replicas broadcast their local transaction ordering correctly, one fragment per round; while Byzantine replicas may broadcast arbitrary sequences in an attempt to manipulate the final transaction ordering. We assume an authenticated communication model in which Byzantine replicas cannot forge messages on behalf of correct replicas.

The system tolerates Byzantine faults under the condition $\mathbf{n} > \frac{(2\gamma+1)\mathbf{f}}{2\gamma-1}$, where γ is a configurable system parameter satisfying $0.5 < \gamma \leq 1$. Intuitively, γ controls the fraction of ordering information required to enforce fairness constraints. When $\gamma = 1$, the requirement reduces to the classical requirement $\mathbf{n} > 3\mathbf{f}$.

6.2.2. Underlying Consensus Layer. Since this work focuses on the transaction ordering algorithm after local orderings are committed, we assume the existence of an underlying Byzantine fault-tolerant consensus layer. This layer commits local ordering fragments proposed by replicas for the ordering layer. It is treated as a black box and required to satisfy the following properties.

- (1) **Round-based operation.** The consensus protocol proceeds in logical rounds indexed by $r = 1, 2, \dots$. Each round produces a consensus decision.
- (2) **Consensus on local ordering fragments.** In each round r , replicas propose local transaction ordering fragments correctly. The consensus layer commits ordering fragments from at least $\mathbf{n} - \mathbf{f}$ replicas every round, and the same set of committed fragments is delivered to every correct replica.
- (3) **Sequential inclusion of correct replicas' proposals.** For every correct replica, all of its local ordering fragments are eventually included in the consensus decisions, in the strict order in which the transactions are observed by the correct replica.
- (4) **Consistency of transaction visibility among correct replicas.** If a transaction T appears in the local ordering of any correct replica, then it eventually appears in the local ordering of all correct replicas, and all the correct local ordering fragments containing T are committed within at most c rounds, for some constant c .

These assumptions abstract away the details of the underlying consensus mechanism and capture the minimal guarantees required by the ordering layer. Given these guarantees, the ordering layer is responsible for producing a final transaction ordering that satisfies γ -Batch-Order-Fairness.

6.3. Problem Statement

6.3.1. Pairwise Precedence Relation. Our goal is to derive a single global transaction ordering that is *fair* with respect to the local orderings of correct replicas. The term “no later” in Definition 6.0.1 is underspecified. In fact, it implies that a pairwise precedence relation may be non-transitive, and that the maximal sets of transactions mutually reachable under this relation form cycles, which we refer to as “batches”.

We model the pairwise precedence relations using a global transaction precedence comparator

$$\text{cmp}_{\mathcal{T}} : \mathcal{T} \times \mathcal{T} \rightarrow \{\text{True}, \text{False}\},$$

For distinct transactions $T \neq T'$, the value $\text{cmp}_{\mathcal{T}}(T, T') = \text{True}$ indicates that fairness requires T to be ordered before T' ; similarly, $\text{cmp}_{\mathcal{T}}(T, T') = \text{False}$ indicates that fairness requires T' to be ordered before T . The comparator induces a *tournament*: for distinct $T, T' \in \mathcal{T}$, $\text{cmp}_{\mathcal{T}}(T, T')$ and

$\text{cmp}_{\mathcal{T}}(T', T)$ are negations of each other, so the precedence between any pair is unambiguously determined. Transitivity, however, is not required, and therefore precedence cycles may arise.

Round-by-round revelation. During consensus protocol execution, the ordering layer does not necessarily know the precedence relation for every transaction pair, because local ordering fragments are committed round by round by the underlying consensus protocol. For a transaction pair, the precedence relation may temporarily remain undetermined until more local ordering fragments are committed.

Instead, committed local orderings up to round r induce a round-specific partial comparator

$$\widehat{\text{cmp}}_{\mathcal{T}}^{(r)} : \mathcal{T} \times \mathcal{T} \rightarrow \{\text{True}, \text{False}\},$$

which captures the precedence relations that have become supported by the local ordering fragments committed up to round r .

We assume monotone refinement: once a precedence relation between a transaction pair becomes supported by committed evidence, it is fixed and will not be revised in later rounds. Formally, if for some round r , $\widehat{\text{cmp}}_{\mathcal{T}}^{(r)}(T, T')$ is defined, then for all $r' > r$, $\widehat{\text{cmp}}_{\mathcal{T}}^{(r')}(T, T') = \widehat{\text{cmp}}_{\mathcal{T}}^{(r)}(T, T')$.

Accordingly, we assume that the global comparator is eventually complete. And the value for each pair is defined by the first round in which the pair becomes determined. Formally, for any transaction pair (T, T') , let r^* be the smallest round such that $\widehat{\text{cmp}}_{\mathcal{T}}^{(r^*)}(T, T')$ is defined. Then we define $\text{cmp}_{\mathcal{T}}(T, T') = \widehat{\text{cmp}}_{\mathcal{T}}^{(r^*)}(T, T')$.

6.3.2. Comparator-Based Formulation of γ -Batch-Order-Fairness.

Comparator-Based Batches. Given the global comparator $\text{cmp}_{\mathcal{T}} : \mathcal{T} \times \mathcal{T} \rightarrow \{\text{True}, \text{False}\}$, we define a directed relation over transactions as follows:

$$T \rightarrow T' \quad \text{iff} \quad \text{cmp}_{\mathcal{T}}(T, T') = \text{True}.$$

Let $\rightsquigarrow$ denote reachability under this relation, i.e., $T \rightsquigarrow T'$ if there exists a directed path from T to T' . Then we define an equivalence relation $\sim$ over $\mathcal{T}$ by

$$T \sim T' \quad \text{iff} \quad (T \rightsquigarrow T') \wedge (T' \rightsquigarrow T).$$

A *batch* is an equivalence class of $\mathcal{T}$ under $\sim$; equivalently, within a batch there exists a directed path from any transaction to any other transaction.

DEFINITION 6.3.1 (Comparator-Based γ -Batch-Order-Fairness). *Let FO be the final ordering over $\mathcal{T}$. We say that FO satisfies comparator-based γ -Batch-Order-Fairness if the following conditions hold:*

- (1) **Linear Batch Partition.** *There exist $k \in \mathbb{N}$ and a partition $\mathcal{T} = B_1 \sqcup B_2 \sqcup \dots \sqcup B_k$ into non-empty contiguous batches such that FO enumerates the elements of B_1 , then B_2 , ..., then B_k (within each B_i in some order), and*

$$\forall 1 \leq i < j \leq k, \forall T \in B_i, \forall T' \in B_j : \text{cmp}_{\mathcal{T}}(T, T') = \text{True}.$$

- (2) **γ -Precedence Respect.** *Let $\text{batch} : \mathcal{T} \rightarrow \{1, 2, \dots, k\}$ map each transaction to the index of the batch containing it. For any $T, T' \in \mathcal{T}$, if at least a γ -fraction of correct replicas receive T before T' , then $\text{batch}(T) \leq \text{batch}(T')$.*

6.3.3. A Sufficient Condition for γ -Batch-Order-Fairness. Although the comparator-based γ -Batch-Order-Fairness provides a characterization of fairness, its definition permits arbitrary ordering within each batch. To enable a cleaner lower-bound argument, we introduce a stronger pairwise precedence property that is sufficient to guarantee γ -Batch-Order-Fairness with respect to a γ -induced comparator.

DEFINITION 6.3.2 (γ -induced comparator). *The comparator $\text{cmp}_{\mathcal{T}}$ is γ -induced if, for any $T, T' \in \mathcal{T}$, at least a γ -fraction of correct replicas receiving T before T' implies $\text{cmp}_{\mathcal{T}}(T, T') = \text{True}$.*

DEFINITION 6.3.3 (Adjacent-Comparator Consistency). *Let FO be the final ordering over $\mathcal{T}$. We say that FO satisfies Adjacent-Comparator Consistency (with respect to $\text{cmp}_{\mathcal{T}}$) if for every position i with $1 \leq i < |\mathcal{T}|$, the comparator on adjacent transactions equals True:*

$$\text{cmp}_{\mathcal{T}}(\text{FO}[i], \text{FO}[i + 1]) = \text{True}.$$

THEOREM 6.3.4 (Sufficiency of Adjacent-Comparator Consistency). *Assume the global comparator $\text{cmp}_{\mathcal{T}} : \mathcal{T} \times \mathcal{T} \rightarrow \{\text{True}, \text{False}\}$ is γ -induced. Let FO be the final ordering over $\mathcal{T}$. If FO satisfies Adjacent-Comparator Consistency, then FO satisfies comparator-based γ -Batch-Order-Fairness.*

PROOF. *Preliminary.* By Adjacent-Comparator Consistency, it holds that $\text{FO}[i] \rightarrow \text{FO}[i+1]$ for every i . Hence for any indices $a < b$, there exists a directed path $\text{FO}[a] \rightarrow \text{FO}[a+1] \rightarrow \dots \rightarrow \text{FO}[b]$, so $\text{FO}[a] \rightsquigarrow \text{FO}[b]$.

(1) *Linear batch partition (by contradiction).* We first show that contiguous batches form a linear partition of FO and that cross-batch comparator directions are forward.

Batch contiguity. Suppose for contradiction that some batch is not contiguous in FO. Then there exist indices $a < b < c$ such that $\text{FO}[a] \sim \text{FO}[c]$ but $\text{FO}[b] \not\sim \text{FO}[a]$. From the preliminary fact, $\text{FO}[a] \rightsquigarrow \text{FO}[b]$ and $\text{FO}[b] \rightsquigarrow \text{FO}[c]$. Also $\text{FO}[c] \rightsquigarrow \text{FO}[a]$ since $\text{FO}[a] \sim \text{FO}[c]$. Thus $\text{FO}[b] \rightsquigarrow \text{FO}[a]$, and together with $\text{FO}[a] \rightsquigarrow \text{FO}[b]$ we obtain $\text{FO}[a] \sim \text{FO}[b]$, contradicting $\text{FO}[b] \not\sim \text{FO}[a]$. Therefore FO can be linearly partitioned into contiguous batches.

Cross-batch direction. Consider any two batches B_i and B_j with $i < j$. Take any positions a in B_i and b in B_j (so $a < b$ in FO). Suppose for contradiction that $\text{cmp}_{\mathcal{T}}(\text{FO}[a], \text{FO}[b]) = \text{False}$. By antisymmetry, this implies $\text{cmp}_{\mathcal{T}}(\text{FO}[b], \text{FO}[a]) = \text{True}$, i.e., $\text{FO}[b] \rightarrow \text{FO}[a]$, hence $\text{FO}[b] \rightsquigarrow \text{FO}[a]$. But from the preliminary fact, $\text{FO}[a] \rightsquigarrow \text{FO}[b]$. Thus $\text{FO}[a] \sim \text{FO}[b]$, meaning they belong to the same batch, contradicting that a is in B_i and b is in B_j with $i < j$. Therefore, for any $i < j$, for all $T \in B_i$ and $T' \in B_j$, it must hold that $\text{cmp}_{\mathcal{T}}(T, T') = \text{True}$.

(2) *γ -precedence respect (by contradiction).* Let $T, T' \in \mathcal{T}$ and suppose at least a γ -fraction of correct replicas receive T before T' . Since $\text{cmp}_{\mathcal{T}}$ is γ -induced, we have $\text{cmp}_{\mathcal{T}}(T, T') = \text{True}$, i.e., $T \rightarrow T'$, and hence $T \rightsquigarrow T'$. Suppose for contradiction that $\text{batch}(T) > \text{batch}(T')$, i.e., $T \in B_j$ and $T' \in B_i$ for some $i < j$. By the linear batch partition established above, T' occupies an earlier position than T in FO; then by the preliminary fact, $T' \rightsquigarrow T$. Combined with $T \rightsquigarrow T'$, this gives $T \sim T'$, forcing T and T' to lie in the same batch—a contradiction. Therefore $\text{batch}(T) \leq \text{batch}(T')$.

Summary. Parts (1) and (2) establish, respectively, the two clauses of the comparator-based γ -Batch-Order-Fairness definition: FO admits a contiguous batch partition $\mathcal{T} = B_1 \sqcup B_2 \sqcup \dots \sqcup B_k$ with forward cross-batch comparator directions (Linear Batch Partition), and the induced index function batch satisfies $\text{batch}(T) \leq \text{batch}(T')$ whenever at least a γ -fraction of correct replicas receive T before T' (γ -Precedence Respect). Hence FO satisfies comparator-based γ -Batch-Order-Fairness, as required.

□

6.3.4. Lower Bound for Adjacent-Comparator Consistency. Enforcing the property Adjacent-Comparator Consistency requires constructing a final ordering whose adjacent transaction pairs respect the global comparator. When the global comparator is transitive, this requirement is equivalent to producing the unique sorted order of the transactions. Since transitive comparators form a subset of all tournament comparators, the worst comparator for enforcing Adjacent-Comparator Consistency is at least as hard as sorting under a transitive comparator, which requires $\Omega(|\mathcal{T}| \log |\mathcal{T}|)$ comparisons, as proved below.

THEOREM 6.3.5 (Expected lower bound in the worst-case over comparators). *Let A be any comparison-based algorithm that, given a transaction set $\mathcal{T}$ together with oracle access to a tournament comparator $\text{cmp}_{\mathcal{T}}$ (i.e., for all $T \neq T'$ in $\mathcal{T}$, exactly one of $\text{cmp}_{\mathcal{T}}(T, T')$ and $\text{cmp}_{\mathcal{T}}(T', T)$ equals *True*), outputs a final ordering FO of $\mathcal{T}$ satisfying Adjacent-Comparator Consistency:*

$$\text{cmp}_{\mathcal{T}}(\text{FO}[i], \text{FO}[i + 1]) = \text{True} \quad \text{for all } i \in \{1, \dots, |\mathcal{T}| - 1\}.$$

Assume the input arrival order is a uniformly random permutation $\pi \sim \text{Unif}(S_{\mathcal{T}})$, where $S_{\mathcal{T}}$ denotes the set of all $|\mathcal{T}|!$ permutations of $\mathcal{T}$, and let $T_A(\text{cmp}_{\mathcal{T}}, \pi)$ be the number of comparator queries that A makes on input $(\text{cmp}_{\mathcal{T}}, \pi)$. Then

$$\max_{\text{cmp}_{\mathcal{T}}} \mathbb{E}_{\pi} [T_A(\text{cmp}_{\mathcal{T}}, \pi)] = \Omega(|\mathcal{T}| \log |\mathcal{T}|).$$

PROOF. It suffices to exhibit one tournament comparator under which A 's expected query count is $\Omega(|\mathcal{T}| \log |\mathcal{T}|)$, since the maximum over all comparators is at least this value.

Fix the *transitive* tournament comparator $\text{cmp}_{\mathcal{T}}^*$ induced by a total order $T_1 < T_2 < \dots < T_{|\mathcal{T}|}$ on $\mathcal{T} = \{T_1, \dots, T_{|\mathcal{T}|}\}$:

$$\text{cmp}_{\mathcal{T}}^*(T_i, T_j) = \text{True} \iff i < j.$$

Under $\text{cmp}_{\mathcal{T}}^*$, Adjacent-Comparator Consistency forces $\text{FO}[i] < \text{FO}[i + 1]$ for every i , and by transitivity the output ordering is uniquely determined:

$$\text{FO} = (T_1, T_2, \dots, T_{|\mathcal{T}|}).$$

Hence, on input $(\text{cmp}_{\mathcal{T}}^*, \pi)$, algorithm A must produce the sorted order of $\mathcal{T}$, and each comparator query $\text{cmp}_{\mathcal{T}}^*(T_i, T_j)$ is equivalent to the standard “ $T_i < T_j$?” comparison. Thus A instantiates a comparison-based sorting algorithm on the input permutation π .

By the classical information-theoretic lower bound for comparison-based sorting, any such algorithm performs at least $\log_2(|\mathcal{T}|!) = \Omega(|\mathcal{T}| \log |\mathcal{T}|)$ comparisons in expectation under a uniformly random input permutation. Therefore

$$\mathbb{E}_{\pi}[T_A(\text{cmp}_{\mathcal{T}}^*, \pi)] = \Omega(|\mathcal{T}| \log |\mathcal{T}|),$$

and taking the maximum over all tournament comparators yields

$$\max_{\text{cmp}_{\mathcal{T}}} \mathbb{E}_{\pi}[T_A(\text{cmp}_{\mathcal{T}}, \pi)] \geq \mathbb{E}_{\pi}[T_A(\text{cmp}_{\mathcal{T}}^*, \pi)] = \Omega(|\mathcal{T}| \log |\mathcal{T}|). \quad \square$$

COROLLARY 6.3.1. *Any comparison-based algorithm that enforces Adjacent-Comparator Consistency makes $\Omega(|\mathcal{T}| \log |\mathcal{T}|)$ expected comparator queries in the worst case over comparators. Consequently, any comparison-based enforcement of γ -Batch-Order-Fairness that relies on Adjacent-Comparator Consistency has the same lower bound.*

6.4. FEAT: AVL-Tree-Based Fair-Ordering Algorithm

In this section, we present FEAT, an AVL-tree-based fair-ordering algorithm. Unlike the previous tournament-construction algorithm [35, 92, 99, 100] that explicitly performs comparison for every transaction pair, FEAT performs comparisons only as needed to determine the final ordering.

FEAT sits in the *fair-ordering layer* of the two-layer architecture for fair transaction ordering protocols shown in Figure 6.1. In each round, the consensus layer commits local-ordering fragments; the fair-ordering layer, which FEAT occupies, uses these commits to produce a fair final ordering. FEAT is therefore a drop-in replacement for the fair-ordering component of existing protocols, leaving the choice of underlying consensus protocol orthogonal to the fairness guarantee. Some prior protocols, e.g., THEMIS [99], place the *fair-ordering layer* before the *consensus layer* rather than after; the fair-ordering algorithm in that layer can still be replaced by FEAT, since FEAT is independent of the order of the two layers.

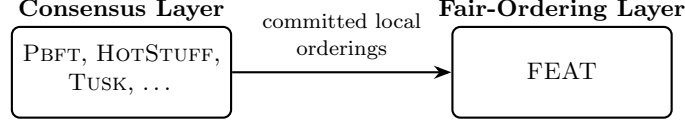


FIGURE 6.1. Two-layer architecture for fair-ordering protocols. Each round, the consensus layer (e.g., PBFT) commits local-ordering fragments; the fair-ordering layer, FEAT and algorithms in prior works (e.g., THEMIS and POMPE) uses these commits to generate a fair final ordering.

In Section 6.5, we prove that FEAT satisfies *Adjacent-Comparator Consistency* and incurs $O(|\mathcal{T}| \log |\mathcal{T}|)$ per-round comparison complexity, where $\mathcal{T}$ denotes the set of transactions inserted into the round's AVL tree. Combined with the lower bound established earlier, this yields a tight $\Theta(|\mathcal{T}| \log |\mathcal{T}|)$ per-round comparison complexity.

6.4.1. Building the AVL Tree of Transactions.

Pairwise weights. For each round r , let $W_r(T_i, T_j)$ denote the number of committed local orderings (up to and including round r) that place transaction T_i before T_j . As additional local ordering fragments are committed, these weights are monotonically non-decreasing.

Weight-based comparator. Given a decidability threshold $\tau_{\text{dec}} > 0$, we define a round-indexed comparator $\text{pcmp}_{\mathcal{T}}^{(r)} : \mathcal{T} \times \mathcal{T} \rightarrow \{\text{True}, \text{False}, \perp\}$ as follows for distinct transactions $T_i \neq T_j$:

- (1) If $W_r(T_i, T_j) < \tau_{\text{dec}}$ and $W_r(T_j, T_i) < \tau_{\text{dec}}$, then it is *undecidable*, i.e., $\text{pcmp}_{\mathcal{T}}^{(r)}(T_i, T_j) = \perp$.
- (2) If $W_r(T_i, T_j) > W_r(T_j, T_i)$ and $W_r(T_i, T_j) \geq \tau_{\text{dec}}$, then $\text{pcmp}_{\mathcal{T}}^{(r)}(T_i, T_j) = \text{True}$.
- (3) If $W_r(T_i, T_j) < W_r(T_j, T_i)$ and $W_r(T_j, T_i) \geq \tau_{\text{dec}}$, then $\text{pcmp}_{\mathcal{T}}^{(r)}(T_i, T_j) = \text{False}$.
- (4) If both weights exceed the threshold and are equal, a deterministic hash function is used as a tie-breaker.

Since the underlying consensus layer guarantees that each transaction eventually appears in the committed local orderings of at least $\mathbf{n} - \mathbf{f}$ correct replicas, we set the decidability threshold to $\tau_{\text{dec}} = \frac{\mathbf{n} - \mathbf{f}}{2}$, which will eventually be reached.

When calculating $\text{pcmp}_{\mathcal{T}}^{(r)}(T_i, T_j)$ in round r , the protocol first checks whether the precedence between (T_i, T_j) has already been decided in some previous round. If so, the previously determined result is returned directly, and the current weights are ignored. Only if the pair has not been decided before does the comparator calculate the round- r weights to determine the outcome. This requirement ensures that once a precedence relation is established, it is never revised in later rounds, which aligns with our description of the global comparator $\text{cmp}_{\mathcal{T}}$ in Section 6.3.1.

Algorithm 3 BUILD AVL TREE IN ROUND(r)

Require: Round index r ; committed local orderings up to each round.

Ensure: Round ordering RO_r obtained from the in-order traversal.

```
1:  $NEW\_SOLID \leftarrow \{T : T \text{ is newly solid in round } r\}$ 
2:  $NEW\_SHADED \leftarrow \{T : T \text{ is newly shaded in round } r\}$ 
3:  $\rho \leftarrow \text{NULL}$ 
4:  $DEFERRED \leftarrow []$ 
5: for all  $T \in NEW\_SOLID$  do
6:    $\rho \leftarrow \text{INSERT}(\rho, T, r)$ 
7: for all  $T \in NEW\_SHADED$  do
8:    $ok \leftarrow \text{TRYINSERT}(\rho, T, r)$ 
9:   if  $ok = \text{False}$  then
10:     $DEFERRED.APPEND(T)$ 
11:  $r' \leftarrow r$ 
12: while  $|DEFERRED| > 0$  do
13:    $r' \leftarrow r' + 1$ 
14:    $WAITFORCOMMIT(r')$ 
15:    $DEFERRED' \leftarrow []$ 
16:   for all  $T \in DEFERRED$  do
17:     $ok \leftarrow \text{TRYINSERT}(\rho, T, r')$ 
18:    if  $ok = \text{False}$  then
19:       $DEFERRED'.APPEND(T)$ 
20:    $DEFERRED \leftarrow DEFERRED'$ 
21:  $RO_r \leftarrow \text{INORDERTRAVERSAL}(\rho)$ 
22: return  $RO_r$ 
```

Algorithm 4 TRYINSERT(ρ, T, r)

Require: Current AVL root ρ ; transaction T ; round index r .

Ensure: True if insertion succeeds; False if aborted due to $\perp$.

```
1:  $u \leftarrow \rho$ 
2: while  $u \neq \text{NULL}$  do
3:    $T_u \leftarrow u.TXN$ 
4:    $q \leftarrow \text{pcmp}_{\mathcal{T}}^{(r)}(T, T_u)$ 
5:   if  $q = \text{True}$  then
6:      $u \leftarrow u.LEFT$ 
7:   else if  $q = \text{False}$  then
8:      $u \leftarrow u.RIGHT$ 
9:   else
10:    return False
11:  $\rho \leftarrow \text{ATTACHLEAFANDREBALANCE}(\rho, T)$ 
12: return True
```

Transaction classification. Each transaction T is assigned a score $\text{score}(T)$, defined as the number of correct replicas whose committed local orderings include T . Recall that a pairwise precedence

relation becomes decidable once the supporting weight exceeds the threshold $\tau_{\text{dec}} = \frac{\mathbf{n}-\mathbf{f}}{2}$. We set the classification thresholds as $\tau_{\text{shaded}} = \tau_{\text{dec}}, \tau_{\text{solid}} = 2 \tau_{\text{dec}}$. Transactions are classified as:

- *solid*: $\text{score}(T) \geq \tau_{\text{solid}}$;
- *shaded*: $\tau_{\text{shaded}} \leq \text{score}(T) < \tau_{\text{solid}}$;
- *blank*: otherwise.

Intuitively, shaded transactions have accumulated sufficient appearance to potentially participate in ordering, while solid transactions have gathered enough support to guarantee decidability against any other transaction.

AVL-tree construction. We generate the final ordering by first constructing an AVL-tree with an empty root ρ in each round r , where the relative position of two transactions is determined by the comparator $\text{pcmp}_{\mathcal{T}}^{(r)}$. We insert only the transactions that are newly classified as non-blank (i.e., solid or shaded) in the current round r . Transactions that were classified as non-blank in earlier rounds are excluded. The complete per-round procedure is shown as Algorithm 3, and TRYINSERT, the per-transaction insertion subroutine for *shaded transactions*, is given as Algorithm 4.

- (1) **Insert solid transactions.** We first insert all solid transactions. Since solid transactions are decidable against all other nodes, each insertion succeeds.
- (2) **Comparator-guided search.** To insert a transaction T , we traverse the tree from the root. At a visited node storing transaction T_u , we query $q \leftarrow \text{pcmp}_{\mathcal{T}}^{(r)}(T, T_u)$. If $q = \text{True}$, we proceed to the left subtree; otherwise ($q = \text{False}$), we proceed to the right subtree.
- (3) **Attach and rebalance.** Once a null child pointer is reached, T is attached as a new leaf. We then retrace the search path, update node heights, and perform standard AVL rotations to restore balance.
- (4) **Round-by-round deferral and retries for shaded transactions.** Shaded transactions are inserted using the same procedure. However, the pairwise precedence between two shaded transactions may be undecidable due to insufficient committed ordering information. If, during search, the comparator returns $\perp$, the insertion of a shaded transaction T is aborted and deferred. At the end of a subsequent round r' , newly committed local orderings update the weights $W_{r'}$, potentially resolving previously undecidable comparisons. Deferred shaded transactions are retried in later rounds until insertion succeeds.

Note on insertion order. Beyond the requirement that solid transactions be inserted before shaded ones, FEAT makes no further assumption about the order in which the newly non-blank transactions of a round are inserted into the AVL tree: any deterministic ordering of the solids, followed by any deterministic ordering of the shadeds, shared by all correct replicas, suffices. In particular, every correct replica may apply the same permutation within each class—e.g., the lexicographic order on transaction identifiers, or a random permutation derived from a global common coin—so that the resulting AVL trees, in-order traversals RO_r , and final ordering FO remain identical across replicas (cf. Theorem 6.5.4). The per-round comparison bound established in Section 6.5 holds for any such permutation.

6.4.2. Finalizing Transaction Ordering. After inserting all new non-blank transactions in round r , an in-order traversal of the AVL tree produces the round ordering RO_r . In Section 6.5, we prove that RO_r satisfies *Adjacent-Comparator Consistency* (Lemma 6.5.0.1).

However, while this property holds *within* each round, it is not automatically guaranteed *across* different rounds. To ensure global consistency of the final transaction ordering, we maintain a *global tail list*, denoted by `TAILLIST`, which carries over non-finalized transactions across rounds.

Global tail list and round stitching. Algorithm 5 gives the full stitching and finalization procedure described in this subsection. At the beginning of round r , `TAILLIST` contains all transactions that have not yet been finalized in previous rounds. After computing the round ordering RO_r , we merge `TAILLIST` and RO_r using the global comparator $\text{cmp}_{\mathcal{T}}$. Since $\text{cmp}_{\mathcal{T}}$ is defined by the earliest round in which a pair becomes decidable, any comparator outcome used during stitching is irrevocable.

Comparator-guided stitching (two-pointer merge). We perform a two-pointer merge. Let i point to the head of `TAILLIST`, and j point to the head of RO_r . We iteratively construct a merged list MERGED_r as follows:

- (1) If i reaches the end of `TAILLIST`, append the remaining suffix of RO_r to MERGED_r and terminate.
- (2) If j reaches the end of RO_r , append the remaining suffix of `TAILLIST` to MERGED_r and terminate.
- (3) Otherwise, let $x \leftarrow \text{TAILLIST}[i]$ and $y \leftarrow RO_r[j]$.
 - If $\text{cmp}_{\mathcal{T}}(x, y) = \text{True}$, append x to MERGED_r and increment i .

Algorithm 5 STITCHANDFINALIZE($r, RO_r, TAILLIST, FO$)

Require: Round index r ; round ordering RO_r ; global tail list $TAILLIST$; global final ordering FO .

Ensure: Updated ($TAILLIST, FO$).

```
1:  $MERGED_r \leftarrow []$ ;  $i \leftarrow 1$ ;  $j \leftarrow 1$ 
2: while  $i \leq |TAILLIST|$  and  $j \leq |RO_r|$  do
3:    $x \leftarrow TAILLIST[i]$ ;  $y \leftarrow RO_r[j]$ 
4:   if  $cmp_{\mathcal{T}}(x, y) = \text{True}$  then
5:      $MERGED_r.APPEND(x)$ ;  $i \leftarrow i + 1$ 
6:   else if  $cmp_{\mathcal{T}}(y, x) = \text{True}$  then
7:      $MERGED_r.APPEND(y)$ ;  $j \leftarrow j + 1$ 
8:   else
9:     wait until more local ordering information is committed in subsequent rounds
10:    continue
11: if  $i > |TAILLIST|$  then
12:   while  $j \leq |RO_r|$  do
13:      $MERGED_r.APPEND(RO_r[j])$ ;  $j \leftarrow j + 1$ 
14: else if  $j > |RO_r|$  then
15:   while  $i \leq |TAILLIST|$  do
16:      $MERGED_r.APPEND(TAILLIST[i])$ ;  $i \leftarrow i + 1$ 
17:  $\ell \leftarrow \text{LASTSOLIDINDEX}(MERGED_r, r)$  ▷Returns 0 if no such transaction exists
18: if  $\ell \geq 1$  then
19:   for  $t \leftarrow 1$  to  $\ell$  do
20:      $FO.APPEND(MERGED_r[t])$ 
21:  $TAILLIST \leftarrow MERGED_r[\ell + 1 ..]$ 
22: return ( $TAILLIST, FO$ )
```

- Else if $cmp_{\mathcal{T}}(y, x) = \text{True}$, append y to $MERGED_r$ and increment j .
- Otherwise (i.e., the order between x and y has not yet been decided by round r), we defer the merge and restart it until more ordering information is committed and additional comparator outcomes are available in a subsequent round.

Finalization and tail update. After constructing $MERGED_r$, we output transactions to the global final ordering FO sequentially from its head, until reaching the last transaction that is classified as *solid* by the end of round r . All transactions up to and including this last solid transaction are permanently appended to FO .

The remaining suffix of $MERGED_r$ becomes the updated $TAILLIST$, which will be stitched with RO_{r+1} in the next round.

Figure 6.2 illustrates a run of FEAT on a 7-transaction example, including the round-by-round deferral and retry mechanism for shaded transactions whose pairwise precedence has not yet become decidable. In this run, the in-order traversal of the final AVL tree is $RO_{r'} = (T_1, T_2, T_3, T_7, T_4, T_5, T_6)$,

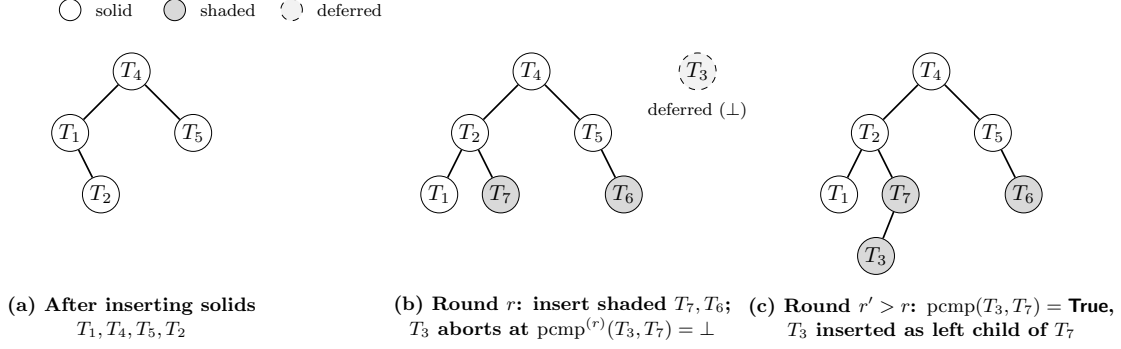


FIGURE 6.2. Execution of FEAT on a 7-transaction example, with insertion order $T_1, T_4, T_5, T_2, T_7, T_3, T_6$ and the latter three transactions classified as *shaded*. We assume that by the end of round r , every pairwise precedence among the transactions is decidable *except* the precedence between T_3 and T_7 : that pair has not yet accumulated enough committed local orderings, so $\text{pcmp}_{\mathcal{T}}^{(r)}(T_3, T_7) = \perp$. **(a)** Solid transactions are inserted first; each insertion succeeds because a solid transaction is decidable against every other transaction. **(b)** Within round r , shaded T_7 and T_6 are inserted; T_3 's insertion is aborted at T_7 because the comparator returns $\perp$, so T_3 is moved to the deferred queue. **(c)** In a subsequent round $r' > r$, newly committed local orderings resolve $\text{pcmp}(T_3, T_7) = \text{True}$; T_3 is successfully re-inserted as the left child of T_7 . The final in-order traversal $T_1, T_2, T_3, T_7, T_4, T_5, T_6$ satisfies Adjacent-Comparator Consistency, and with a total of 13 pairs compared queries: $T_1 : \emptyset$, $T_4 : \{T_1\}$, $T_5 : \{T_1, T_4\}$, $T_2 : \{T_4, T_1\}$, $T_7 : \{T_4, T_1, T_2\}$, $T_6 : \{T_4, T_1\}$, $T_3 : \{T_4, T_2, T_7\}$.

in which T_6 is shaded and falls after the last solid transaction T_5 . Consequently, the finalization step of round r appends only the prefix up to and including T_5 to the global final ordering FO, while T_6 is carried into TAILLIST to be stitched with RO_{r+1} in the next round.

6.5. Correctness and Complexity Proof

We prove correctness and complexity tightness of FEAT. First, we show that each round ordering RO_r satisfies Adjacent-Comparator Consistency (Lemma 6.5.0.1), that stitching preserves this property (Lemma 6.5.0.2), and that consistency holds across round boundaries (Lemma 6.5.0.3), implying that the final ordering FO is globally Adjacent-Comparator Consistent (Theorem 6.5.1). Second, we prove that the global comparator in FEAT is γ -induced (Lemma 6.5.1.1), which together with the Adjacent-Comparator Consistency yields γ -Batch-Order-Fairness (Theorem 6.5.2). Finally, we establish that FEAT achieves $O(|\mathcal{T}| \log |\mathcal{T}|)$ per-round comparison complexity (Theorem 6.5.3) and that this bound is tight (Corollary 6.5.1). We additionally show that a fair-ordering

protocol with FEAT satisfies safety such that every correct replica produces the same final ordering FO (Theorem 6.5.4).

LEMMA 6.5.0.1 (Adjacent-Comparator Consistency of round ordering RO_r in FEAT). *For each round r , the in-order traversal ordering RO_r satisfies Adjacent-Comparator Consistency: for all i , it holds that $\text{cmp}_{\mathcal{T}}(\text{RO}_r[i], \text{RO}_r[i+1]) = \text{True}$.*

PROOF. We prove the claim by induction on k , the number of inserted transaction nodes.

Base case. For $k = 0$ or $k = 1$, the tree has at most one node, and $\text{RO}_r^{(k)}$ has length at most one. Hence there are no adjacent pairs, and the claim holds trivially.

Induction hypothesis. Assume that after inserting k nodes (and performing all necessary AVL rebalancing), the in-order traversal $\text{RO}_r^{(k)}$ satisfies $\forall i, \text{cmp}_{\mathcal{T}}(\text{RO}_r^{(k)}[i], \text{RO}_r^{(k)}[i+1]) = \text{True}$.

Induction step. Consider inserting the $(k+1)$ -st key x into the AVL tree rooted at ρ_k , using the binary-search-tree (BST) branching rule induced by $\text{cmp}_{\mathcal{T}}$, followed by AVL rebalancing. Let ρ_{k+1} denote the root of the resulting AVL tree, and let $\text{RO}^{(k+1)} := \text{In}(\rho_{k+1})$ be its in-order traversal.

We first examine the moment immediately after inserting x and before AVL-tree rebalancing. At this moment, x is inserted as a *leaf*, and thus the sequence $\text{RO}_r^{(k+1)}$ is obtained from $\text{RO}_r^{(k)}$ by inserting the single new element x at exactly one position. Consequently, all adjacent pairs in $\text{RO}_r^{(k+1)}$ remain unchanged from $\text{RO}_r^{(k)}$ except those involving x . Let p and s denote the in-order predecessor and successor of x in $\text{RO}_r^{(k+1)}$ (if they exist), that is, p immediately precedes x and s immediately follows x in the sequence $\text{RO}_r^{(k+1)}$.

Predecessor. If p exists, then x , as a leaf node, lies in the right subtree of p in the tree before rebalancing. Hence the insertion search path for x must have visited the node storing p and taken the right branch at p . By the BST branching rule induced by cmp , this implies $\text{cmp}_{\mathcal{T}}(p, x) = \text{True}$.

Successor. If s exists, then x lies in the left subtree of s in the tree before rebalancing. Thus the insertion search path must have visited the node storing s and taken the left branch at s . By the branching rule, this implies $\text{cmp}_{\mathcal{T}}(x, s) = \text{True}$.

All other adjacent pairs in $\text{RO}_r^{(k+1)}$ are identical to those in $\text{RO}_r^{(k)}$ and therefore satisfy the property by the induction hypothesis. Finally, AVL-tree rebalancing consists of a constant number of local rotations, each of which preserves the in-order traversal sequence. Hence rebalancing

does not change $RO_r^{(k+1)}$, and the *Adjacent-Comparator Consistency* property remains true after inserting the $(k+1)$ -st node. This completes the induction. $\square$

LEMMA 6.5.0.2 (Stitching preserves Adjacent-Comparator consistency in FEAT). *Fix a round r . Let RO_r be the round ordering produced by the in-order traversal of the AVL tree in round r , and let $TAILLIST$ be the tail list carried into round r . Let $MERGED_r$ be the output of the comparator-guided stitching procedure (two-pointer merge) that merges $TAILLIST$ and RO_r using $\text{cmp}_{\mathcal{T}}$. Then $MERGED_r$ satisfies Adjacent-Comparator Consistency:*

$$\forall t, \quad \text{cmp}_{\mathcal{T}}(\text{MERGED}_r[t], \text{MERGED}_r[t+1]) = \text{True}.$$

PROOF. We prove by considering every adjacent pair in $MERGED_r$ and showing that the consistency holds.

Invariant. At any time during the merge, if the last element appended to $MERGED_r$ is denoted by z , then for both current heads $TAILLIST[i]$ and $RO_r[j]$, whenever they exist, it holds that

$$i \leq |TAILLIST| \Rightarrow \text{cmp}_{\mathcal{T}}(z, TAILLIST[i]) = \text{True},$$

$$j \leq |RO_r| \Rightarrow \text{cmp}_{\mathcal{T}}(z, RO_r[j]) = \text{True}.$$

We establish this invariant as follows: If the algorithm appends $TAILLIST[i]$, it must be because $\text{cmp}_{\mathcal{T}}(TAILLIST[i], RO_r[j]) = \text{True}$. Moreover, since $TAILLIST$ already satisfies Adjacent-Comparator Consistency, we have the new last element $TAILLIST[i]$ precedes the new tail head $TAILLIST[i+1]$, i.e., $\text{cmp}_{\mathcal{T}}(TAILLIST[i], TAILLIST[i+1]) = \text{True}$. The case where the algorithm appends $RO_r[j]$ is symmetric, using the fact that RO_r satisfies Adjacent-Comparator Consistency (Lemma 6.5.0.1). Therefore, the invariant holds throughout the merge.

Adjacent pairs in $MERGED_r$. Consider any adjacent pair $(\text{MERGED}_r[t], \text{MERGED}_r[t+1])$. Element $\text{MERGED}_r[t+1]$ is appended immediately after $\text{MERGED}_r[t]$ in some merge iteration. At that iteration, $\text{MERGED}_r[t+1]$ is one of the current heads of the two lists. By the invariant, immediately before appending $\text{MERGED}_r[t+1]$, the last element in $MERGED_r$ (which is exactly $\text{MERGED}_r[t]$) satisfies $\text{cmp}_{\mathcal{T}}(\text{MERGED}_r[t], \text{MERGED}_r[t+1]) = \text{True}$. Hence every adjacent pair in $MERGED_r$ is comparator-consistent.

This proves that $MERGED_r$ satisfies Adjacent-Comparator Consistency. $\square$

LEMMA 6.5.0.3 (Boundary consistency between FO_{r-1} and MERGED_r). *Let FO_{r-1} be the global final ordering after round $r - 1$, and let TAILLIST_{r-1} be the tail list carried into round r . Let z be the last transaction of FO_{r-1} (if any), and let m be the head transaction of MERGED_r (if any). If both lists are non-empty, then $\text{cmp}_{\mathcal{T}}(z, m) = \text{True}$.*

PROOF. We consider two cases.

Case 1: m is the head transaction of TAILLIST_{r-1} . Since TAILLIST_{r-1} is exactly the non-finalized suffix of the previous round's merged list, the head transaction TAILLIST_{r-1} was the immediate successor of z at the end of round $r - 1$. By Lemma 6.5.0.2, the previous merged list satisfies Adjacent-Comparator Consistency, and in particular the boundary pair between the last finalized transaction z and the head m of the merged list is consistent, i.e., $\text{cmp}_{\mathcal{T}}(z, m) = \text{True}$.

Case 2: m is the head transaction of RO_r . At the end of round $r - 1$ the transaction m was still *blank* (i.e., not yet present as a non-blank transaction), otherwise it would have been included either in FO_{r-1} or in TAILLIST_{r-1} . Since m is *blank* and z is *solid* by the end of round $r - 1$, we have $\text{score}(z) \geq \tau_{\text{solid}}$ and $\text{score}(m) < \tau_{\text{shaded}}$, so the local orderings of more than $\tau_{\text{solid}} - \tau_{\text{shaded}} = \tau_{\text{dec}}$ correct replicas have committed z but not m by the end of round $r - 1$. Thus, $W_{r-1}(z, m) > \tau_{\text{dec}} = \tau_{\text{shaded}} > W_{r-1}(m, z)$, hence $\text{cmp}_{\mathcal{T}}(z, m) = \text{True}$.

Therefore, in both cases we have $\text{cmp}_{\mathcal{T}}(z, m) = \text{True}$, which completes the proof. $\square$

THEOREM 6.5.1 (Adjacent-Comparator Consistency of Final Ordering in FEAT). *The final transaction ordering FO produced by FEAT satisfies Adjacent-Comparator Consistency, i.e.,*

$$\forall t, \quad \text{cmp}_{\mathcal{T}}(\text{FO}[t], \text{FO}[t + 1]) = \text{True}.$$

PROOF. We prove the claim by showing that throughout the execution of FEAT, whenever transactions are appended to the global final ordering FO , the *Adjacent-Comparator Consistency* property is preserved for the updated global final ordering.

Induction over rounds. Initially, FO is empty, and hence Adjacent-Comparator Consistency holds trivially.

Assume inductively that Adjacent-Comparator Consistency holds for FO_{r-1} . By Lemma 6.5.0.2, MERGED_r constructed in round r satisfies Adjacent-Comparator Consistency. By Lemma 6.5.0.3,

the boundary pair between the last element of FO_{r-1} and the first element of MERGED_r is also comparator-consistent.

In round r , the updated final ordering FO_r is obtained by appending a prefix of MERGED_r to FO_{r-1} . Since the appended prefix is a contiguous segment of MERGED_r , all adjacent pairs within it are consistent; and since the boundary pair with FO_{r-1} is consistent, the concatenation preserves Adjacent-Comparator Consistency.

Therefore, FO_r satisfies Adjacent-Comparator Consistency. By induction over r , the final transaction ordering FO produced by FEAT satisfies Adjacent-Comparator Consistency. $\square$

LEMMA 6.5.1.1 (FEAT comparator is γ -induced). *Fix any two distinct transactions $T_i \neq T_j$. If at least a γ -fraction of correct replicas receive T_i before T_j , then the global comparator used by FEAT satisfies $\text{cmp}_{\mathcal{T}}(T_i, T_j) = \text{True}$.*

PROOF. We prove by contradiction. Assume that at least a γ -fraction of correct replicas receive T_i before T_j , but nevertheless $\text{cmp}_{\mathcal{T}}(T_i, T_j) = \text{False}$. By the definition of $\text{cmp}_{\mathcal{T}}$ (via the first round in which the pair becomes decidable), the event $\text{cmp}_{\mathcal{T}}(T_i, T_j) = \text{False}$ implies that there exists some round r such that the opposite direction reaches the decidability threshold, i.e., $W_r(T_j, T_i) \geq \tau_{\text{dec}}$. In FEAT, $\tau_{\text{dec}} = \frac{\mathbf{n}-\mathbf{f}}{2}$.

Since at least a γ -fraction of correct replicas receive T_i before T_j , at most a $(1 - \gamma)$ -fraction of correct replicas can receive T_j before T_i . Moreover, Byzantine replicas (at most $\mathbf{f}$) may contribute arbitrarily to the committed local orderings. Therefore, for every round r ,

$$W_r(T_j, T_i) \leq (1 - \gamma)(\mathbf{n} - \mathbf{f}) + \mathbf{f}.$$

Combining with $W_r(T_j, T_i) \geq \tau_{\text{dec}} = \frac{\mathbf{n}-\mathbf{f}}{2}$ yields

$$(1 - \gamma)(\mathbf{n} - \mathbf{f}) + \mathbf{f} \geq \frac{\mathbf{n} - \mathbf{f}}{2}.$$

After rearrangement, we get

$$\mathbf{n} \leq \frac{(2\gamma + 1)\mathbf{f}}{2\gamma - 1}.$$

This contradicts the system model requirement $\mathbf{n} > \frac{(2\gamma+1)\mathbf{f}}{2\gamma-1}$. Hence our assumption is false, and we conclude that $\text{cmp}_{\mathcal{T}}(T_i, T_j) = \text{True}$.

This proves that the global comparator $\text{cmp}_{\mathcal{T}}$ is γ -induced. $\square$

THEOREM 6.5.2 (FEAT achieves γ -Batch-Order-Fairness). *The final transaction ordering FO produced by FEAT satisfies γ -Batch-Order-Fairness with respect to comparator $\text{cmp}_{\mathcal{T}}$.*

PROOF. By Lemma 6.5.1.1, the global comparator $\text{cmp}_{\mathcal{T}}$ used by FEAT is γ -induced. By Theorem 6.5.1, the final transaction ordering FO produced by FEAT satisfies Adjacent-Comparator Consistency. Therefore, the conditions of Theorem 6.3.4 (Sufficiency of Adjacent-Comparator Consistency) hold. Applying Theorem 6.3.4, we conclude that FO satisfies γ -Batch-Order-Fairness with comparator $\text{cmp}_{\mathcal{T}}$. $\square$

THEOREM 6.5.3 (FEAT per-round upper bound). *Fix a round r . Assume that the number of non-blank transactions that participate in round r (i.e., the transactions inserted into the round- r AVL tree, and thus appearing in RO_r) is $O(|\mathcal{T}|)$. Then the number of comparator queries against $\text{cmp}_{\mathcal{T}}$ made by FEAT in round r is $O(|\mathcal{T}| \log |\mathcal{T}|)$.*

PROOF. In round r , FEAT inserts the $O(|\mathcal{T}|)$ non-blank transactions into an AVL tree. By standard AVL-tree properties, an AVL tree containing $O(|\mathcal{T}|)$ nodes has height $O(\log |\mathcal{T}|)$. Therefore, inserting a transaction follows a root-to-leaf search path of length $O(\log |\mathcal{T}|)$ and thus incurs $O(\log |\mathcal{T}|)$ comparator queries.

Although shaded transactions may be retried when the comparator returns $\perp$, the system model guarantees that for each transaction, within at most c rounds (for a constant c), the committed local orderings of all correct replicas will include this transaction. Consequently, any transaction first attempted in round $r' \leq r$ can be successfully inserted by round $r' + c$, with $O(c)$ attempts in total. Each insertion attempt follows a root-to-leaf path of length $O(\log |\mathcal{T}|)$, so the total number of comparator queries spent on the AVL-tree of round r is $O(|\mathcal{T}| \log |\mathcal{T}|)$.

This within- c -rounds visibility assumption is essential for the per-round bound. Without it, a shaded transaction whose pairwise precedence against some other tree node never becomes decidable would be deferred and retried indefinitely, so the per-round comparison count grows without bound. Even under the strictly weaker retry policy of re-inserting a shaded transaction only when a newly committed local ordering contains it, each shaded transaction may still be retried $O(\mathbf{n})$ times in the worst case, again exceeding the $O(|\mathcal{T}| \log |\mathcal{T}|)$ budget.

In the same round, FEAT stitches the round ordering RO_r with the persistent TAILLIST using the two-pointer merge. Let $a := |\text{TAILLIST}|$ and $b := |RO_r|$. Each merge step performs exactly one comparator query between the two current head elements and advances at least one pointer. Hence the total number of comparator queries during stitching in round r is at most $a + b$. Under the assumption that the number of participating non-blank transactions in each round is $O(|\mathcal{T}|)$, we have both $a = O(|\mathcal{T}|)$ and $b = O(|\mathcal{T}|)$. Therefore, the stitching phase contributes $O(|\mathcal{T}|)$ comparator queries in round r , which is asymptotically dominated by $O(|\mathcal{T}| \log |\mathcal{T}|)$.

Combining the two parts, the total number of comparator queries performed by FEAT in round r is $O(|\mathcal{T}| \log |\mathcal{T}|)$. $\square$

COROLLARY 6.5.1 (Tight per-round comparison bound). *Fix a round r and assume that the number of non-blank transactions that participate in round r is $\Theta(|\mathcal{T}|)$. Then FEAT has $\Theta(|\mathcal{T}| \log |\mathcal{T}|)$ expected comparison complexity in round r in the worst case over comparators.*

PROOF. The upper bound $O(|\mathcal{T}| \log |\mathcal{T}|)$ follows from Theorem 6.5.3. For the matching lower bound, consider any comparison-based algorithm that outputs a final ordering satisfying Adjacent-Comparator Consistency. By Theorem 6.3.5, the lower bound is $\Omega(|\mathcal{T}| \log |\mathcal{T}|)$. Since FEAT is comparison-based and enforces Adjacent-Comparator Consistency within each round, this lower bound applies to FEAT as well. Combining the upper and lower bounds yields $\Theta(|\mathcal{T}| \log |\mathcal{T}|)$ expected comparison complexity in the worst case over comparators. $\square$

THEOREM 6.5.4 (Safety). *Assume (i) the consensus layer commits the same local-ordering fragments to every correct replica each round, and (ii) all correct replicas apply the same deterministic transaction-insertion order within each non-blank class (cf. the note at the end of Section 6.4). Then every correct replica running FEAT produces the same final ordering FO.*

PROOF. By the system model (Section 6.2), in each round r the consensus layer commits the same set of local-ordering fragments to every correct replica. Every step performed by FEAT in round r is a deterministic function of the committed orderings up to round r and of FO and TAILLIST carried from the previous round.

We prove by induction on r that at the end of round r , every correct replica holds an identical AVL tree, identical round ordering RO_r , identical TAILLIST, and identical FO_r . The base case $r = 0$ is trivial, since TAILLIST and FO_0 are initially empty at every replica. For the inductive step, both

the committed input from the consensus layer to round $r + 1$ and the per-replica state inherited from round r are identical across correct replicas; applying the deterministic round procedure of FEAT to identical inputs yields identical outputs. Hence FO_{r+1} , TAILLIST_{r+1} , and the AVL tree are identical at every correct replica. Taking the limit as $r \rightarrow \infty$, the final ordering FO is the same at every correct replica. $\square$

6.6. Conclusion

In this chapter, we resolved the per-round comparison complexity of enforcing γ -Batch-Order-Fairness by modeling fairness constraints as a global pairwise comparator over transactions. We introduced *Adjacent-Comparator Consistency* as a sufficient condition for γ -Batch-Order-Fairness with respect to a γ -Induced Comparator, and proved a lower bound of $\Omega(|\mathcal{T}| \log |\mathcal{T}|)$ comparator queries in the worst case over comparators via reduction from comparison-based sorting. We then presented FEAT, an AVL-tree-based fair-ordering algorithm that matches this bound by assembling the final ordering through balanced-BST insertions. The matching upper and lower bounds pin the per-round complexity at $\Theta(|\mathcal{T}| \log |\mathcal{T}|)$, and FEAT serves as a drop-in replacement for the fair-ordering layer of existing BFT fair-ordering protocols.

CHAPTER 7

Conclusion

This dissertation has presented five contributions toward building high-performance and fair Byzantine consensus protocols. We conclude by summarizing these contributions and discussing directions for future research.

7.1. Summary of Contributions

Our work addresses the key limitations of rotational consensus protocols along two dimensions: *performance* and *fairness*.

SpotLess: Concurrent Rotational Consensus with Rapid View Synchronization. We proposed SpotLess, a rotational consensus protocol that combines the high throughput of concurrent consensus architectures with the simplicity of streamlined rotational consensus. SpotLess introduces a Rapid View Synchronization mechanism that synchronizes replicas every view with only $O(\Delta)$ worst-case latency, without relying on costly threshold signatures in most cases. Experimental results demonstrate that SpotLess outperforms PBFT by up to 430%, Narwhal by up to 137%, and HotStuff by up to 3803%, while maintaining stable latency even during failures.

HotStuff-1: Linear Consensus with One-Phase Speculation. We introduced HotStuff-1, the first speculative BFT protocol with linear communication complexity. By incorporating speculation into HotStuff-2, HotStuff-1 allows replicas to send speculative responses to clients one phase earlier, reducing client-perceived latency. We identified and resolved the Prefix Speculation Dilemma through the Prefix Speculation Rule and the No-Gap Rule. Furthermore, HotStuff-1 employs a slotting mechanism that allows correct leaders to propose multiple blocks per view, mitigating the impact of slow or Byzantine leaders and enhancing throughput.

Carry: Tail-Forking Resilience with Linear Complexity. We proposed Carry, a practical drop-in mechanism for HotStuff-like protocols that resolves tail-forking attacks while preserving linear communication complexity. Carry treats replica votes as first-class citizens that transport knowledge of the last safe block forward, preventing Byzantine leaders from skipping safe blocks and

enabling correct leaders to reinstate previous safe blocks. Carry-Tail, our full atomic broadcast solution combining Carry with HotStuff-2, achieves efficient linearity and strong performance under both malicious leadership and benign transient delays.

FairDAG: Fair Transaction Ordering over DAG-Based Consensus. We presented FairDAG, a fair-ordering framework that runs fairness protocols atop multi-proposer DAG-based consensus protocols. By leveraging the multi-proposer design and causal structure of DAG protocols, FairDAG removes the single-leader bottleneck present in prior fair-ordering protocols, constrains Byzantine replicas’ ability to manipulate transaction ordering, and reduces the requirement on the number of correct replicas. Our two variants, FairDAG and FairDAG-RL, guarantee Ordering Linearizability and γ -batch-order-fairness respectively, outperforming prior protocols in both throughput and fairness quality under normal and adversarial conditions.

FEAT: Fair and Efficient Adversarial Transaction Ordering. We introduced FEAT, which establishes the complexity-theoretic foundation for fair transaction ordering under Byzantine settings. We proved that enforcing Adjacent-Comparator Consistency, a sufficient condition for γ -Batch-Order-Fairness, incurs a worst-case lower bound of $\Omega(|\mathcal{T}| \log |\mathcal{T}|)$ comparisons. FEAT achieves this tight bound via an AVL-tree-based construction, improving upon the quadratic complexity of all prior fair-ordering protocols.

7.2. Future Directions

While this dissertation has made progress toward high-performance and fair Byzantine consensus, several promising directions remain.

Unifying Performance and Fairness. Our work has advanced performance and fairness as two separate dimensions. A natural next step is to integrate these techniques into a unified protocol that simultaneously achieves high throughput, low latency, resilience to tail-forking attacks, and fair transaction ordering. For instance, combining the concurrent consensus architecture of SpotLess or the slotting mechanism of HotStuff-1 with the fair-ordering framework of FairDAG could yield a protocol that excels along both dimensions.

Fairness under Stronger Adversarial Models. The fair-ordering protocols in this dissertation assume that Byzantine replicas constitute fewer than one-third of the total. Investigating fairness

guarantees under stronger adversarial models, such as adaptive adversaries who can corrupt replicas dynamically, remains an important open problem.

Practical Deployment and Real-World Evaluation. While our protocols have been evaluated through prototype implementations, deploying them in production blockchain systems introduces additional challenges, including heterogeneous network conditions, varying workloads, and integration with smart contract execution engines. Bridging the gap between protocol design and practical deployment is a critical direction for future work.

Bibliography

- [1] Tpc-c benchmark: Standard specification. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf, 2010. Accessed: 2025-01-16.
- [2] Apache resilientdb (incubating), 2024.
- [3] Spacecoin blue paper. <https://github.com/spacecoinxyz/research/blob/main/publications/Blue-Paper-Spacecoinxyz.pdf>, 2024.
- [4] Ittai Abraham, Guy Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR*, *abs/1803.05069*, 2018.
- [5] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance, 2017.
- [6] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.
- [7] Ittai Abraham and Gilad Stern. Information theoretic hotstuff. In *International Conference on Principles of Distributed Systems*, 2020.
- [8] Mark Abspoel, Thomas Attema, and Matthieu Rabaud. Malicious security comes for free in consensus with leaders. *Cryptology ePrint Archive*, 2020.
- [9] M AdelsonVelskii and Evgenii Mikhailovich Landis. An algorithm for the organization of information. Technical report, 1962.
- [10] Amit Agarwal, Kushal Babel, Sourav Das, and Babak Poorebrahim Gilkalaye. Time-lock encrypted storage for blockchains. *Cryptology ePrint Archive*, 2025.
- [11] Orestis Alpos, Ignacio Amores-Sesar, Christian Cachin, and Michelle Yeo. Eating Sandwiches: Modular and Lightweight Elimination of Transaction Reordering Attacks. In *27th International Conference on Principles of Distributed Systems (OPODIS 2023)*, volume 286 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:22. Schloss Dagstuhl, 2024.
- [12] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. CAPER: A cross-application permissioned blockchain. *Proc. VLDB Endow.*, 12(11):1385–1398, 2019.
- [13] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *SIGMOD '21: International Conference on Management of Data*, pages 76–88. ACM, 2021.

- [14] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The bedrock of Byzantine fault tolerance: A unified platform for BFT protocols analysis, implementation, and experimentation. In Laurent Vanbever and Irene Zhang, editors, *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, pages 371–400. USENIX Association, 2024.
- [15] Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. Leaderless consensus. In *41st IEEE International Conference on Distributed Computing Systems*, pages 392–402. IEEE, 2021.
- [16] Apache Software Foundation. Apache ResilientDB (Incubating), 2023.
- [17] Claudio A Ardagna, Marco Anisetti, Barbara Carminati, Ernesto Damiani, Elena Ferrari, and Christian Rondanini. A blockchain-based trustworthy certification process for composite services. In *2020 IEEE International Conference on Services Computing (SCC)*, pages 422–429. IEEE, 2020.
- [18] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. Shoal++: High throughput dag bft can be fast! *arXiv preprint arXiv:2405.20488*, 2024.
- [19] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. A fair consensus protocol for transaction ordering. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 55–65. IEEE, 2018.
- [20] Aditya Asgaonkar. Proposer LMD Score Boosting, Ethereum Consensus-Specs., 2021.
- [21] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quéma, and Marko Vukolic. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, 2015.
- [22] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: Redundant Byzantine fault tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306. IEEE, 2013.
- [23] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency DAG consensus with fast commit path. *CoRR*, abs/2310.14821, 2023.
- [24] Jeb Bearer, Benedikt Bünz, Philippe Camacho, Binyi Chen, Ellie Davidson, Ben Fisch, Brendon Fish, Gus Gutoski, Fernando Krell, Chengyu Lin, et al. The espresso sequencing network: Hotshot consensus, tiramisu data-availability, and builder-exchange. *Cryptology ePrint Archive*, 2024.
- [25] Christian Berger and Hans P. Reiser. Scaling Byzantine consensus: A broad analysis. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pages 13–18. ACM, 2018.
- [26] Adithya Bhat, Akhil Bandrupalli, Manish Nagaraj, Saurabh Bagchi, Aniket Kate, and Michael K. Reiter. EESMR: energy efficient BFT - SMR for the masses. In *Proceedings of the 24th International Middleware Conference, Middleware 2023, Bologna, Italy, December 11-15, 2023*, pages 1–14. ACM, 2023.
- [27] Erik-Oliver Blass and Florian Kerschbaum. BOREALIS: building block for sealed bid auctions on blockchains. In *ASIA CCS ’20: The 15th ACM Asia Conference on Computer and Communications Security*, pages 558–571. ACM, 2020.

- [28] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.
- [29] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
- [30] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.
- [31] Felix Brandt, Vincent Conitzer, Ulle Endriss, Jérôme Lang, and Ariel D. Procaccia. *Handbook of Computational Social Choice*. Cambridge University Press, Cambridge, UK, 2016.
- [32] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.
- [33] Vitalik Buterin. Ethereum white paper: A next-generation smart contract and decentralized application platform. <https://ethereum.org/en/whitepaper/>, 2013.
- [34] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.
- [35] Christian Cachin, Jovana Mićić, Nathalie Steinhauer, and Luca Zanolini. Quick order fairness. In *International Conference on Financial Cryptography and Data Security*, pages 316–333. Springer, 2022.
- [36] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS’05)*, pages 191–201. IEEE, 2005.
- [37] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In *31st International Symposium on Distributed Computing*, volume 91, pages 1:1–1:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.
- [38] Miguel Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, Massachusetts Institute of Technology, 2001.
- [39] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.
- [40] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.
- [41] Junchao Chen, Alberto Sonnino, Lefteris Kokoris-Kogias, and Mohammad Sadoghi. Thunderbolt: Causal concurrent consensus and execution. *arXiv preprint arXiv:2407.09409*, 2024.
- [42] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. 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)*, volume 246 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:21. Schloss Dagstuhl, 2022.
- [43] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Byzantine consensus is $\theta(n^2)$: the dolev-reischuk bound is tight even in partial synchrony! *Distributed Comput.*, 37:89–119, 2023.

- [44] Marquis de Condorcet. *Essay on the Application of Analysis to the Probability of Majority Decisions*. Imprimerie Royale, Paris, 1785.
- [45] Andrei Constantinescu, Diana Ghinea, Lioba Heimbach, Zilin Wang, and Roger Wattenhofer. A fair and resilient decentralized clock network for transaction ordering. *arXiv preprint arXiv:2305.05206*, 2023.
- [46] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154. ACM, 2010.
- [47] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.
- [48] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *ArXiv*, abs/1904.05234, 2019.
- [49] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50. ACM, 2022.
- [50] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, pages 123–140. ACM, 2019.
- [51] Diem. DiemBFT consensus protocol, 2020.
- [52] Tien Tuan Anh Dinh, Dakai Kang, and Mohammad Sadoghi. FEAT: Fair and efficient adversarial transaction ordering. In *Proceedings of the 2026 ACM Symposium on Principles of Distributed Computing (PODC ’26)*, PODC ’26, New York, NY, USA, July 2026. Association for Computing Machinery.
- [53] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Trans. Knowl. Data Eng.*, 30(7):1366–1385, 2018.
- [54] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1085–1100. ACM, 2017.
- [55] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
- [56] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A shared database on blockchains. *Proc. VLDB Endow.*, 12(11):1597–1609, 2019.
- [57] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- [58] Flow. Flow: The blockchain for open worlds. <https://flow.com>, 2025. Accessed: 2025-01-21.

- [59] Pranav Garimidi, Joachim Neu, and Max Resnick. Multiple concurrent proposers: Why and how. *arXiv preprint arXiv:2509.23984*, 2025.
- [60] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. In *International conference on financial cryptography and data security*, pages 296–315. Springer, 2022.
- [61] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nikolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP*, pages 51–68. ACM, 2017.
- [62] Neil Girdharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. No-commit proofs: Defeating livelock in BFT, 2021.
- [63] Neil Girdharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 1–23, 2024.
- [64] Neil Girdharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 1–23, 2024.
- [65] Neil Girdharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. BeeGees: Stayin’ alive in chained BFT. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC ’23*, page 233–243, New York, NY, USA, 2023. Association for Computing Machinery.
- [66] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019.
- [67] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. Scalable byzantine reliable broadcast. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.
- [68] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous BFT protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818. ACM, 2020.
- [69] Suyash Gupta. *Resilient and Scalable Architecture for Permissioned Blockchain Fabrics*. PhD thesis, University of California, Davis, USA, 2021.
- [70] Suyash Gupta, Mohammad Javad Amiri, and Mohammad Sadoghi. Chemistry behind agreement. In *13th Conference on Innovative Data Systems Research, CIDR*. www.cidrdb.org, 2023.

- [71] Suyash Gupta, Jelle Hellings, Sajjad Rahnema, and Mohammad Sadoghi. An in-depth look of BFT consensus in blockchain: Challenges and opportunities. In *Proceedings of the 20th International Middleware Conference Tutorials, Middleware*, pages 6–10. ACM, 2019.
- [72] Suyash Gupta, Jelle Hellings, Sajjad Rahnema, and Mohammad Sadoghi. Blockchain consensus unraveled: virtues and limitations. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, pages 218–221. ACM, 2020.
- [73] Suyash Gupta, Jelle Hellings, Sajjad Rahnema, and Mohammad Sadoghi. Building high throughput permissioned blockchain fabrics: Challenges and opportunities. *Proc. VLDB Endow.*, 13(12):3441–3444, 2020.
- [74] Suyash Gupta, Jelle Hellings, Sajjad Rahnema, and Mohammad Sadoghi. Proof-of-Execution: Reaching consensus through fault-tolerant speculation. In Yannis Velegrakis, Demetris Zeinalipour-Yazti, Panos K. Chrysanthis, and Francesco Guerra, editors, *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, pages 301–312. OpenProceedings.org, 2021.
- [75] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. Brief announcement: Revisiting consensus protocols through wait-free parallelization. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146, pages 44:1–44:3. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.
- [76] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. *Fault-Tolerant Distributed Transactions on Blockchain*. Synthesis Lectures on Data Management. Morgan & Claypool, 2021.
- [77] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 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*, pages 1392–1403. IEEE, 2021.
- [78] Suyash Gupta, Dakai Kang, Dahlia Malkhi, and Mohammad Sadoghi. Brief announcement: Carry the tail in consensus protocols. In *39th International Symposium on Distributed Computing (DISC 2025)*, pages 59–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.
- [79] Suyash Gupta, Dakai Kang, Dahlia Malkhi, and Mohammad Sadoghi. Carry the tail in consensus protocols. *arXiv preprint arXiv:2508.12173*, 2025.
- [80] Suyash Gupta, Sajjad Rahnema, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global scale resilient blockchain fabric. *Proc. VLDB Endow.*, 13(6):868–883, 2020.
- [81] Suyash Gupta, Sajjad Rahnema, Erik Linsenmayer, Faisal Nawab, and Mohammad Sadoghi. Reliable transactions in serverless-edge architecture. In *39th IEEE International Conference on Data Engineering, ICDE 2023*, pages 301–314. IEEE, 2023.
- [82] Suyash Gupta, Sajjad Rahnema, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. Dissecting bft consensus: In trusted components we trust! In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 521–539, 2023.

- [83] Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. Dissecting BFT consensus: In trusted components we trust! In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 521–539. ACM, 2023.
- [84] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. In *40th International Conference on Distributed Computing Systems*, pages 754–764. IEEE, 2020.
- [85] Jelle Hellings, Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. On the correctness of speculative consensus, 2022.
- [86] Jelle Hellings and Mohammad Sadoghi. ByShard: sharding in a Byzantine environment. *The VLDB Journal*, 32(6):1343–1367, 2023.
- [87] 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. Confidential consortium framework: Secure multiparty applications with confidentiality, integrity, and high availability. *Proc. VLDB Endow.*, 17(2):225–240, 2023.
- [88] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.
- [89] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-HotStuff: A fast and robust BFT protocol for blockchains. *IEEE Transactions on Dependable and Secure Computing*, 2023.
- [90] Mohammad Mussadiq Jalalzai and Kushal Babel. Monadbft: Fast, responsive, fork-resistant streamlined consensus. *ArXiv*, abs/2502.20692, 2025.
- [91] Dakai Kang, Junchao Chen, Anh Dinh, and Mohammad Sadoghi. Fairdag, 2025. Accessed: 2025-04-01.
- [92] Dakai Kang, Junchao Chen, Tien Tuan Anh Dinh, and Mohammad Sadoghi. FairDAG: Consensus fairness over multi-proposer causal design. *Proceedings of the VLDB Endowment*, 19(2):194–207, 2025.
- [93] Dakai Kang, Suyash Gupta, Dahlia Malkhi, and Mohammad Sadoghi. Hotstuff-1: Linear consensus with one-phase speculation. *Proceedings of the ACM on Management of Data*, 3(3):1–29, 2025.
- [94] Dakai Kang, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 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, 2024.
- [95] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2nd edition, 2014.
- [96] Alireza Kavousi, Duc V. Le, Philipp Jovanovic, and George Danezis. BlindPerm: Efficient MEV mitigation with an encrypted mempool and permutation. In *29th International Conference on Principles of Distributed Systems (OPODIS 2025)*, LIPIcs. Schloss Dagstuhl, 2026.

- [97] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.
- [98] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. Cordial miners: Fast and efficient consensus for every eventuality. In Rotem Oshman, editor, *37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L'Aquila, Italy*, volume 281 of *LIPIcs*, pages 26:1–26:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.
- [99] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 475–489, 2023.
- [100] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Advances in Cryptology-CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III 40*, pages 451–480. Springer, 2020.
- [101] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *Proceedings of the 25th USENIX Conference on Security Symposium*, pages 279–296. USENIX, 2016.
- [102] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009.
- [103] Lucas Kuhring, Zsolt István, Alessandro Sorniotti, and Marko Vukolić. StreamChain: Building a low-latency permissioned blockchain for enterprise use-cases. In *2021 IEEE International Conference on Blockchain (Blockchain)*, pages 130–139. IEEE, 2021.
- [104] Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 25–36, 2020.
- [105] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, 2001. Distributed Computing Column 5.
- [106] Leslie Lamport. The part-time parliament. In *Concurrency: the works of Leslie Lamport*, pages 277–317. 2019.
- [107] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.
- [108] Andrew Lewis-Pye. Quadratic worst-case message complexity for state machine replication in the partial synchrony model, 2022.
- [109] Andrew Lewis-Pye and Ittai Abraham. Fever: optimal responsive view synchronisation. *arXiv preprint arXiv:2301.09881*, 2023.
- [110] Andrew Lewis-Pye, Dahlia Malkhi, Oded Naor, and Kartik Nayak. Lumiere: Making optimal BFT for partial synchrony practical. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, pages 135–144, 2024.

- [111] Rujia Li, Xuanwei Hu, Qin Wang, Sisi Duan, and Qi Wang. Transaction fairness in blockchains, revisited. *IEEE Transactions on Dependable and Secure Computing*, 2025.
- [112] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. Scalable Byzantine consensus via hardware-assisted secret sharing. *IEEE Trans. Comput.*, 68(1):139–151, 2019.
- [113] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 485–500, USA, 2016. USENIX Association.
- [114] Dumitrel Loghin, Tien Tuan Anh Dinh, Aung Maw, Chen Gang, Yong Meng Teo, and Beng Chin Ooi. Blockchain goes green? part ii: Characterizing the performance and cost of blockchains on the cloud and at the edge, 2022.
- [115] Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-Dumbo transformer: Asynchronous consensus as fast as the pipelined BFT. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2159–2173, 2022.
- [116] Hanzheng Lyu, Shaokang Xie, Jianyu Niu, Ivan Beschastnikh, Yinqian Zhang, Mohammad Sadoghi, and Chen Feng. Orthrus: Accelerating multi-bft consensus through concurrent partial ordering of transactions. *arXiv preprint arXiv:2501.14732*, 2024.
- [117] Mads Frederik Madsen, Mikkel Gaub, Malthe Ettrup Kirkbro, and Søren Debois. Transforming Byzantine faults using a trusted execution environment. In *15th European Dependable Computing Conference*, pages 63–70. IEEE, 2019.
- [118] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.
- [119] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. BBKA-CHAIN: one-message, low latency BFT consensus on a DAG. *CoRR*, abs/2310.06335, 2023.
- [120] Dahlia Malkhi and Pawel Szalachowski. Maximal extractable value (mev) protection on a dag. *arXiv preprint arXiv:2208.00940*, 2022.
- [121] Tejas Mane, Xiao Li, Mohammad Sadoghi, and Mohsen Lesani. Ava: Fault-tolerant reconfigurable geo-replication on heterogeneous clusters. *arXiv preprint arXiv:2412.01999*, 2024.
- [122] Meter.io. Meter: Decentralized finance infrastructure. <https://meter.io>, 2025. Accessed: 2025-01-21.
- [123] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.
- [124] Ke Mu, Bo Yin, Alia Asheralieva, and Xuetao Wei. Separation is good: A faster order-fairness byzantine consensus. 2024.
- [125] Heena Nagda, Shubhendra Pal Singhal, Mohammad Javad Amiri, and Boon Thau Loo. Rashnu: Data-dependent order-fairness. *Proceedings of the VLDB Endowment*, 17(9):2335–2348, 2024.

- [126] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.
- [127] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. 2021.
- [128] Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. *Distributed Computing*, 37(1):19–33, 2024.
- [129] Arvind Narayanan and Jeremy Clark. Bitcoin’s academic pedigree. *Communications of the ACM*, 60(12):36–45, 2017.
- [130] Faisal Nawab and Mohammad Sadoghi. Consensus in data management: From distributed commit to blockchain. *Found. Trends Databases*, 12(4):221–364, 2023.
- [131] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.
- [132] Burak Öz, Benjamin Kraner, Nicolò Vallarano, Bingle Stegmann Kruger, Florian Matthes, and Claudio Juan Tessone. 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*, page 1–8, New York, NY, USA, 2023. Association for Computing Machinery.
- [133] Eunchan Park, Taeung Yoon, Hocheol Nam, Deepak Maram, and Min Suk Kang. On frontrunning risks in batch-order fair systems for blockchains (extended version). Cryptology ePrint Archive, Paper 2025/1168, 2025.
- [134] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.
- [135] Sajjad Rahnama, Suyash Gupta, Thamir M. Qadah, Jelle Hellings, and Mohammad Sadoghi. Scalable, resilient, and configurable permissioned blockchain fabric. *Proc. VLDB Endow.*, 13(12):2893–2896, 2020.
- [136] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. 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*, pages 298–311. OpenProceedings.org, 2022.
- [137] Michael K. Reiter. Secure agreement protocols: reliable and atomic group multicast in rampart. In *Conference on Computer and Communications Security*, 1994.
- [138] Ben Riva, Alberto Sonnino, and Lefteris Kokoris-Kogias. Seahorse: Efficiently mixing encrypted and normal transactions. In *Financial Cryptography and Data Security (FC 2025)*. Springer, 2025.
- [139] Ethereum Roadmap. Proposer-builder separation, 2024.
- [140] Christian Rondanini, Barbara Carminati, Federico Daidone, and Elena Ferrari. Blockchain-based controlled information sharing in inter-organizational workflows. In *2020 IEEE International Conference on Services Computing (SCC)*, pages 378–385. IEEE, 2020.
- [141] Pingcheng Ruan, Tien Tuan Anh Dinh, Qian Lin, Meihui Zhang, Gang Chen, and Beng Chin Ooi. Lineagechain: a fine-grained, secure and efficient data provenance system for blockchains. *VLDB J.*, 30(1):3–24, 2021.

- [142] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. Eactors: Fast and flexible trusted computing using SGX. In Paulo Ferreira and Liuba Shrira, editors, *Proceedings of the 19th International Middleware Conference*, pages 187–200. ACM, 2018.
- [143] Caspar Schwarz-Schilling. Retroactive Proposer Rewards, 2022.
- [144] Caspar Schwarz-Schilling, Fahad Saleh, Thomas Thiery, Jennifer Pan, Nihar Shah, and Barnabé Monnot. Time Is Money: Strategic Timing Games in Proof-Of-Stake Protocols. In *5th Conference on Advances in Financial Technologies (AFT 2023)*, volume 282 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [145] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 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*, pages 955–970. ACM, 2020.
- [146] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1722–1743. ACM, 2021.
- [147] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. Sailfish: Towards improving the latency of DAG-based BFT. *Cryptology ePrint Archive*, Paper 2024/472, 2024.
- [148] Man-Kit Sit, Manuel Bravo, and Zsolt István. 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*, pages 55–65. ACM, 2021.
- [149] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. Shoal: Improving DAG-BFT latency and robustness. *arXiv preprint arXiv:2306.03058*, 2023.
- [150] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2705–2718. ACM, 2022.
- [151] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-BFT: High-throughput BFT for blockchains, 2019.
- [152] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 17–33. ACM, 2022.
- [153] Chrysoula Stathakopoulou, Signe Rüsch, Marcus Brandenburger, and Marko Vukolić. Adding fairness to order: Preventing front-running attacks in bft protocols using tees. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 34–45. IEEE, 2021.

- [154] Xiao Sui, Sisi Duan, and Haibin Zhang. Marlin: Two-phase BFT with linearity. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 54–66, 2022.
- [155] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 1–17, 2021.
- [156] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 1493–1509, 2020.
- [157] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2001.
- [158] Costa Tsaousis and Phil Whineray. FireHOL–Linux firewalling and traffic shaping for humans, 2023.
- [159] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Maarten van Steen, 3th edition, 2017.
- [160] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.
- [161] Suzhen Wu, Zhanhong Tu, Yuxuan Zhou, Zuocheng Wang, Zhirong Shen, Wei Chen, Wei Wang, Weichun Wang, and Bo Mao. FASTSync: a fast delta sync scheme for encrypted cloud storage in high-bandwidth network environments. *ACM Transactions on Storage*, 19(4):1–22, 2023.
- [162] Shaokang Xie, Dakai Kang, Hanzheng Lyu, Jianyu Niu, and Mohammad Sadoghi. Fides: Scalable censorship-resistant dag consensus via trusted components. *arXiv preprint arXiv:2501.01062*, 2025.
- [163] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.
- [164] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019.
- [165] Rui Yuan, Yubin Xia, Haibo Chen, Binyu Zang, and Jan Xie. ShadowEth: Private smart contract on public blockchain. *J. Comput. Sci. Technol.*, 33(3):542–556, 2018.
- [166] Pouriya Zarbafian and Vincent Gramoli. Lyra: Fast and scalable resilience to reordering attacks in blockchains. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 929–939. IEEE, 2023.
- [167] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. Gem²-tree: A gas-efficient structure for authenticated range queries in blockchain. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 842–853. IEEE, 2019.
- [168] Gengrui Zhang, Fei Pan, Sofia Tijanic, and Hans-Arno Jacobsen. PrestigeBFT: Revolutionizing view changes in BFT consensus algorithms with reputation mechanisms. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 1930–1943. IEEE, 2024.

- [169] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 633–649, 2020.