Brief Announcement: Carry the Tail in Consensus Protocols

Suyash Gupta
University of Oregon, Eugene, OR, USA

Dakai Kang
University of California Davis, CA, USA

Dahlia Malkhi
University of California Santa Barbara, CA, USA

Mohammad Sadoghi
University of California Davis, CA, USA

Abstract

We present **Carry-the-Tail**, the first deterministic atomic broadcast protocol in partial synchrony that, after GST, simultaneously guarantees two desirable properties: (i) a constant fraction of commits are proposed by non-faulty leaders against tail-forking attacks, and (ii) optimal, worst-case quadratic communication under a cascade of faulty leaders. The solution also guarantees linear amortized communication, i.e., the steady-state is linear. Combining these two desirable properties was not simultaneously achieved previously: on one hand, prior atomic broadcast solutions achieve per-view linear word communication complexity. However, they face a significant degradation in throughput under tail-forking attack. On the other hand, existing solutions to tail-forking attacks require either quadratic communication steps or computationally-prohibitive SNARK generation.

The key technical contribution is **Carry**, a practical drop-in mechanism for streamlined protocols in the HotStuff family. Carry guarantees good performance against tail-forking and removes most leader-induced stalls, while retaining linear traffic and protocol simplicity. Carry-the-Tail implements the Carry mechanism on HotStuff-2.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Consensus, Blockchain, BFT

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.59

Related Version *Full Version*: https://arxiv.org/abs/2508.12173 [6]

Funding *Mohammad Sadoghi*: This work is partially funded by NSF Award Number 2245373.

1 Introduction

In a "streamlined" Byzantine Fault Tolerant (BFT) consensus approach, which was pioneered by HotStuff [15], the consensus protocol has a simple and uniform structure: each *view* is a single quorum exchange between a leader and voters, and each such exchange carries a new leader proposal. Since a single quorum exchange does not suffice to achieve a consensus decision, leaders have to rely on the next leader to drive a second exchange that commits the previous proposal. HotStuff-like (HS-like) protocols [1, 2, 4, 9, 10, 13, 14, 15] are widely adopted in modern blockchain and decentralized systems for their conceptual simplicity, responsive liveness, linear communication, and censorship resistance.

BeeGees [5] exposed a vulnerability of the streamlined regime. In a *tail-forking* attack, a malicious or sluggish next leader can skip over the previous leader's proposal. Repeated attacks by bad leaders might significantly degrade throughput.

39th International Symposium on Distributed Computing (DISC 2025).
Editor: Dariusz R. Kowalski; Article No. 59; pp. 59:1–59:7
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We present **Carry**, a lightweight, drop-in mechanism for HS-like protocols that defends against tail-forking. The mechanism also boosts performance under straggler leaders by ensuring safe progress with aggressive responsiveness without waiting for full quorums. Notably, Carry incurs a linear communication overhead per view.

Carry is a generic mechanism. We demonstrate its application to HotStuff-2 to create Carry-the-Tail, a full consensus solution that achieves the following two guarantees: ρ-tail-resilience (see Definition 1) and liveness under hostile or sluggish leadership with $O(n)$ communication per view.

Previously, no solution simultaneously achieved both properties. Existing tail-forking defenses either require expensive quadratic communication [5] or computationally heavy SNARKs to prove the absence of quorum certificates [7, 8].

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

2 Overview of Carry-the-Tail

Carry is a generic liveness-boost for protocols in the HotStuff family. The setting for this work is described in the full paper [6]; briefly, the system consists of $N = 3F + 1$ replicas with up to F Byzantine faulty, or F_{actual} faulty in an actual execution, $F_{actual} \leq F$, and partially synchronous communication. A full-fledged protocol referred to as Carry-the-Tail is described here. It integrates the Carry approach into HotStuff-2.

Proposing and Voting

The Carry-the-Tail protocol operates in a view-by-view manner. Each view v has a known designated leader L_v performing a single quorum-exchange with replicas. The leader broadcasts a proposal B_v to the replicas; replicas respond (subject to safety rules) with a NEW-VIEW message carrying a signature-share on B_v. The signature-share is referred to as a *vote* on B_v.

A Quorum-Certificate (QC) consists of a threshold-signature by a quorum of $2F + 1$ replicas. We say that the proposal is *certified* when $\mathsf{QC}(B_v)$ is formed. To form QCs, leaders collect signature-shares for previous proposals and aggregate them. Broadcasting a QC incurs only linear word-communication complexity.

Commit Safety and Liveness

Each proposal B_v includes an opaque payload and meta-information. The meta-information references a history known to the leader. It includes $B_v.QC$ prior to view v, the highest known certified tail. Chaining proposals to one another using cryptographic certificates is utilized along with protocol voting and commit rules to ensure **safety**.

Figure 1 depicts a failure-free flow of the protocol with leader proposals chained to one another via QCs. Upon receiving a proposal B_v, a replica becomes *locked* on $B_v.QC$. Locking is key to ensuring safety: in the future, the replica pledges that it will only accept proposals that extend $B_v.QC$ or a QC from a higher view. The figure shows a failure-free execution snippet with four consecutive leaders chaining proposals one after another.

A *commit* necessitates two consecutive successful views. If a leader L_v forms $\mathsf{QC}(v-1)$ for the proposal B_{v-1} immediately preceding it, then the proposal becomes committed once there are $2F + 1$ signature-shares on B_v, forming $\mathsf{QC}(v)$. Anyone can learn this commit

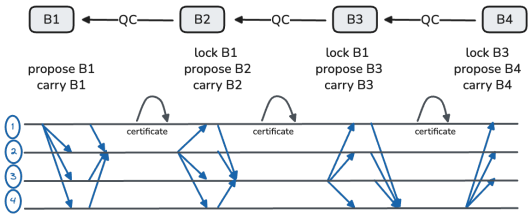

Figure 1 Basic flow in a failure-free scenario.

decision by obtaining these votes. In particular, anyone observing a chain $B_*.QC = \mathsf{QC}(v)$, $B_v.QC = \mathsf{QC}(v-1)$ learns that B_{v-1} has been committed. In Figure 1, B_1 and B_2 have been committed, and anyone observing this chain can learn these decisions.

To guarantee **liveness**, a leader proposes if it received a QC from the immediately-preceding view or a timeout. The timeout is implemented by a separate view-synchronization (Pacemaker) module [3, 11, 12]. Any (linear) Pacemaker can be plugged here. The Pacemaker guarantees that sufficient time has elapsed for all honest replicas to have entered view v and for their NEW-VIEW messages to have arrived at the leader.

The problem with Failed Views and Tail-Forking

We first explain the tail-forking attack which was exposed in BeeGees [5], and later describe how Carry prevents it.

Figure 2(a) depicts a scenario with failed view B_2, zooming in on the transition from B_2 to B_3. In view 2, no replica succeeds to vote for B_2. L_3, the leader of view 3, times out without having formed a QC for B_2 and must skip B_2.

The problem is that a malicious L_3 could exploit this and omit votes for B_2 to wrongfully skip it. Figure 2(b) shows that B_3 could ignore the $2F+1$ honest votes (the dashed arrows) for B_2 and wrongfully skip it. More generally, a view v suffers a *tail-forking* attack if $2F+1$ signature shares are sent by honest replicas on B_v, but the next leader L_{v+1} intentionally ignores and skips it. The attack can be caused by either a malicious or a sluggish leader.

Regardless of the cause for tail-forking, it causes significant performance degradation. First, a proposal (e.g., B_2) is unnecessarily dropped. Second, the latency to a commit decision on pending earlier proposals increases. Moreover, tail-forking might occur frequently if there are many malicious/sluggish leaders. In the worst case, F bad leaders perfectly interspersed among honest views as depicted in Figure 4(perf1), where leaders $L_3, L_5, ...$, are bad. This might cause $O(N)$ throughput degradation and $O(N)$ latency increase.

As explained below, the Carry method prevents the tail-foring attacks from happening.

Protecting the Tail

Because replicas send NEW-VIEW messages to the next leader, in lieu of a vote, they can send a signature-share on an empty vote ("$\perp$"). In Carry, a *justified* skip over a tail B_v is allowed if it is accompanied by an *Empty Certificate (EC)* consisting of a threshold signature by $2F+1$ replicas on $\perp$. An *Empty Certificate* cannot possibly be formed if F+1 honest replicas voted for B_v; but if only F or fewer honest replicas voted for B_v, it could form, which affects neither safety nor liveness. Figure 3(c) depicts shows how *EC*s justifies a skipped tail, where L_3 collects $2F+1$ empty votes (the yellow arrows) for view 2 and forms an $\mathsf{EC}(2)$.

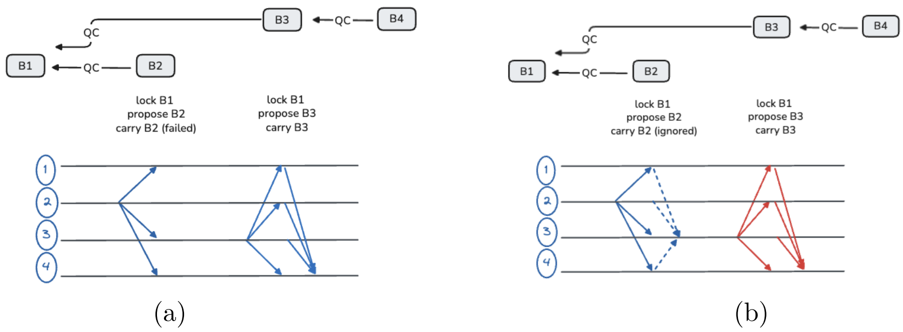

Figure 2 Without tail protection: (a) B_2 failed and benignly skipped. (b) B_2 tail-forked.

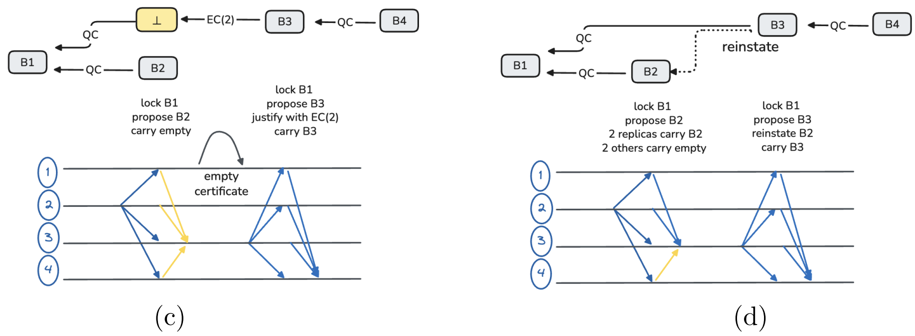

Figure 3 With Carry protection: (c) B_2 failed and EC-justified skipped. (d) B_2 reinstated.

Reinstating Uncertified Tails

If there is neither a QC nor an EC, previous solutions (e.g., [5]) required leaders to justify skipping the tail by using all the votes, resulting in cubic word-communication complexity in the worst case with F consecutive malicious leaders.

Instead, the main idea in Carry is to force the next leader to *reinstate* the last (possibly uncertified) voted block. To implement this, Carry introduces an optional field $B_v.reinstate$. This field references an uncertified earlier proposal B_x that becomes an integral part of B_v. As illustrated in Figure 3(d), L_3 receives $F+1 = 2$ votes for B_2 and two empty votes, preventing the formation of either $\mathsf{QC}(2)$ or $\mathsf{EC}(2)$. Consequently, B_2 is *reinstated* in B_3.

With this new mechanism, as shown in Figure 4(perf2), a leader must justify skipping a tail via an EC or else reinstate the tail. Notably, reinstating also helps if insufficient votes arrive for B_v due to mere sluggishness.

ρ-Tail-Resilience

Reinstating protects the tail in case an EC cannot be formed on an immediately preceding view. Unfortunately, tail-forking of B_v remains possible by having two consecutive malicious leaders following B_v. Attacks by two consecutive malicious leaders, L_{v+1} and L_{v+2}, are depicted in Figure 4(perf3), e.g., L_3 and L_4. First, L_{v+1} "fails" in view $v+1$, not forming $\mathsf{QC}(v)$. Next, B_{v+2} uses $\mathsf{EC}(v+1)$ 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

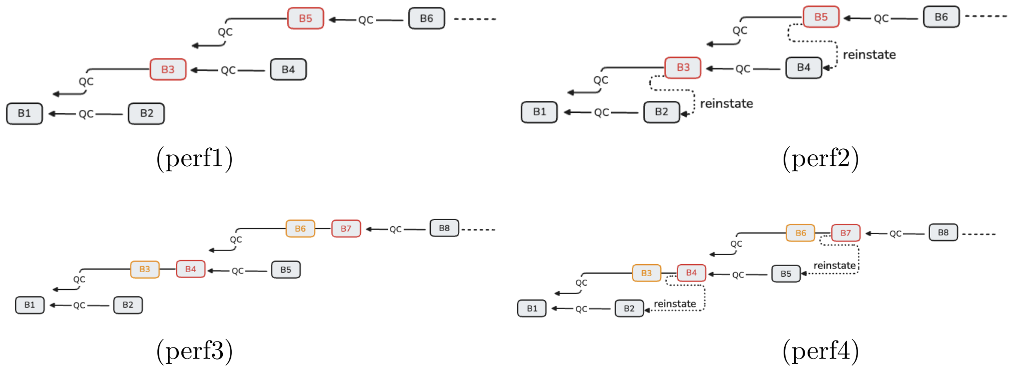

Figure 4 (perf1) Performance Degradation due to Tail-Forking. (perf2) Improvement with Reinstating. (perf3) Degradation under Two-Leader Attacks. (perf4) Improvement with ρ-Tail-Resilience.

bad views. Hence, in each view rotation, all but (F_{actual}/ρ) proposals are protected from tail-forking. This property is referred to as $\rho{-}Tail{-}Resilience$. $2{-}Tail{-}Resilience$ is depicted in Figure 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.

The overhead incurred is $O(\rho N)$ word communication.

Comparison with AHL

BeeGees formulated a property called *"Any-Honest-Leader commit"* (AHL): after GST, once an honest leader proposes in a view, that block will be committed after at most k subsequent honest-leader views[1]. BeeGees employs a complex leader handover in order to satisfy AHL (in one variant, it incurs quadratic word-communication per view, and in a second variant, it relies on computationally-prohibitive SNARK proofs for compressing communication).

The property *ρ-tail-resilience* guarantees, under a reasonably small choice of ρ (e.g. $\rho = 6$), that a large constant fraction of honest leader proposals become committed even against the worst-case tail-forking attacks. It is worth noting that by setting $\rho = f$, we get the same AHL property in Carry as in BeeGees, while incurring the same communication burden. However, arguably Carry has simpler logic and also allows ρ to be a tunable parameter in production.

3 Carry

In this section, we show how Carry is implemented, including the *Empty Certificate Justification* and *Reinstate* mechanisms. Replicas carry their votes (which are possibly empty) for the last ρ views in their NEW-VIEW messages, and the new leader must reinstate the last uncertified proposal.

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

Carry Implementation

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 $\mathsf{QC}(x)$ known to L_v is attached as $B_v.qc = \mathsf{QC}(x)$;

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 $B_v.reinstated = T$;

Justification: If $v - x \leq \rho$, then empty-certificates are attached to B_v for each view between x and v, which B_v does **not** extend. It is worth noting that if $B_v.reinstated$ exists, empty-certificates for views preceding the reinstated block are recursively attached within the chain of reinstated blocks.

Voting Rule. A replica accepts and votes for the proposal B_v from leader L_v of view v if:

1. $B_v.qc$ has a higher or equal view than the replica's *lock*,
2. B_v adheres to the Carry Rule above.

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

3.1 The Carry-the-Tail Protocol

Akin to HotStuff-2, the Carry-the-Tail protocol flows view-by-view. At the end of each view, the replicas and the incoming leader perform a handover protocol as follows [2]:

Replica $\longrightarrow$ 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 votes (possibly empty) in the past ρ views.[3]

Leader $\longrightarrow$ 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,
3. attaches empty-certificates for each view between view(T) and v.

References

1 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.

2 Benjamin Y. Chan and Rafael Pass. Simplex consensus: A simple and fast consensus protocol. *IACR Cryptol. ePrint Arch.*, 2023:463, 2023. URL: https://api.semanticscholar.org/CorpusID:259092405.

[2] See full version [6] for detailed design, pseudocode and correctness proofs.

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

3 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. doi:10.4230/LIPIcs.DISC.2022.14.

4 Diem. DiemBFT consensus protocol, 2020. URL: https://github.com/diem/diem/tree/latest/consensus.

5 Neil Giridharan, 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, pages 233–243, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3583668.3594572.

6 Suyash Gupta, Dakai Kang, Dahlia Malkhi, and Mohammad Sadoghi. Carry the tail in consensus protocols, August 2025. doi:10.48550/arXiv.2508.12173.

7 Mohammad Mussadiq Jalalzai and Kushal Babel. Monadbft: Fast, responsive, fork-resistant streamlined consensus. *ArXiv*, abs/2502.20692, 2025. doi:10.48550/arXiv.2502.20692.

8 Mohammad Mussadiq Jalalzai, Chen Feng, and Victoria Lemieux. Vbft: Veloce byzantine fault tolerant consensus for blockchains. *ArXiv*, abs/2310.09663, 2023. doi:10.48550/arXiv.2310.09663.

9 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. doi:10.1145/3725308.

10 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. doi:10.1109/ICDE60146.2024.00157.

11 Andrew Lewis-Pye. Quadratic worst-case message complexity for state machine replication in the partial synchrony model, 2022. arXiv:2201.01107.

12 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. doi:10.1145/3662158.3662787.

13 Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft, 2023. URL: https://api.semanticscholar.org/CorpusID:259144145.

14 Jianyu Niu, Fangyu Gai, Mohammad M. Jalalzai, and Chen Feng. On the performance of pipelined hotstuff. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021. doi:10.1109/INFOCOM42981.2021.9488706.

15 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019. URL: https://api.semanticscholar.org/CorpusID:197644531.