BeeGees: Stayin' Alive in Chained BFT

Neil Giridharan
UC Berkeley
USA
giridhn@berkeley.edu

Florian Suri-Payer
Cornell University
USA
fsp@cs.cornell.edu

Matthew Ding
UC Berkeley
USA
matthewding@berkeley.edu

Heidi Howard
Microsoft Research
UK
heidi.howard@microsoft.com

Ittai Abraham
VMware Research
Israel
iabraham@vmware.com

Natacha Crooks
UC Berkeley
USA
ncrooks@berkeley.edu

ABSTRACT

Modern *chained* Byzantine Fault Tolerant (BFT) systems leverage a combination of pipelining and leader rotation to obtain both efficiency and fairness. These protocols, however, require a sequence of three or four *consecutive* honest leaders to commit operations. Therefore, even simple leader failures such as crashes can weaken liveness, resulting in high commit latency or lack of commit all together. We show that, unfortunately, this vulnerability is inherent to all existing BFT protocols that rotate leaders with pipelined agreement. To resolve this liveness shortcoming we present *BeeGees*[1], a novel chained BFT protocol that successfully commits blocks even with non-consecutive honest leaders. It does this while also maintaining quadratic word complexity with threshold signatures, linear word complexity with SNARKs, and responsiveness between consecutive honest leaders. BeeGees reduces the expected commit latency of HotStuff by a factor of three under failures, and the worst-case latency by a factor of seven.

CCS CONCEPTS

• **Theory of Computation; Distributed Algorithms**;

KEYWORDS

Consensus, Blockchain, BFT

1 INTRODUCTION

Blockchain systems have emerged as a promising way for mutually distrustful parties to compute over shared data. Byzantine Fault Tolerant (BFT) state machine replication (SMR), the core protocol in most blockchains, provides to applications the abstraction of a centralized, trusted, and always available server. BFT SMR guarantees that a set of replicas will agree on a common sequence of operations, even though some nodes may misbehave. Blockchain systems add two additional constraints over prior work 1) operation ordering should be *fair*: it must closely follow the order in which operations are submitted, and offer no single party undue influence in the process. Protocols without fairness can be abused by the application: participants may censor or front-run to gain economic advantages 2) protocols should maintain low latency and high throughput when scaling to large number of replicas.

To address these concerns, recent BFT protocols targeted at blockchains, such as HotStuff [34], DiemBFT [32], Fast-Hotstuff [21] as well as the largest Proof-of-Stake system, Ethereum (Casper FFG [8]), are structured around two key building blocks:

Chaining. Every BFT protocol requires a (worst-case) minimum of two voting rounds (henceforth *phases*). Each voting phase aims to establish a *quorum certificate (QC)* by collecting a set of signed votes from a majority of honest replicas. Blockchain systems *pipeline* the voting phases of consecutive proposals to avoid redundant coordination and cryptography as well as to minimize the commit latency of subsequent requests: the system can use the quorum certificate of the second voting phase of block i to certify the first phase of block $i+1$. Each block then requires (on average) generating and verifying the signatures of a single QC. This is especially important for large participant sets as QC sizes grow linearly with the number of replicas, increasing cryptographic costs.

Leader-Speaks-Once (LSO). To minimize fairness concerns associated with leader-based solutions and to decrease the influence of adaptive adversaries (who control the network), many BFT protocols targeted at blockchains adopt a *leader-speak-once* (LSO) model. In LSO, each leader proposes and certifies a single block after which the leader is immediately rotated out as part of a new *view*. Electing a different leader per block limits the leader's influence; it can manipulate transactions in the proposed block only. Traditional BFT protocols (such as PBFT [9]), in contrast, adopt a *stable-leader* paradigm in which leaders are only replaced if they fail to make progress through a fallback *view change* protocol. Failures are assumed to be infrequent, and thus protocol complexities (and costs) intentionally move into the view change, allowing for a simpler and more efficient failure-free steady case.

While a joint approach that is both *chained* and *leader-speak-once* (CLSO) is desirable, the combination of these two properties also introduces a new challenge: how to preserve safety when block commitment is spread across leaders? This work observes that all prior CLSO protocols solve this challenge by *unintentionally* relinquishing liveness – and proposes a novel protocol that manages to avoid this trade-off (without sacrificing performance).

The problem. To maintain safety, block commitment in prior CLSO protocols requires a sequence of k QCs in consecutive views

[1] BeeGees *stays (a-)live* against the odds.

PODC '23, June 19–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0121-4/23/06…$15.00
https://doi.org/10.1145/3583668.3594572

(where $k \in \{2,3\}$ depending on protocol). Consequently, liveness is only guaranteed in the presence of $k+1$ *consecutive* honest (non-faulty) leaders. In the remainder of this paper, we refer to this property as CHL (consecutive honest leaders).

Definition 1.1. (CHL). After GST, if an honest leader in view v proposes a value and views $v+1, v+2, ..., v+k$ (contiguous views) have honest leaders, then it is guaranteed to commit the value.

This guarantee introduces significant performance penalties in practice [27]. We show in Section 3 that in HotStuff [34] for example – the protocol at the core of the former Diem blockchain [33] (now Aptos [31]) – a single faulty leader may suffice to prevent *any* block from being committed for some system configurations. Further, we show that even for arbitrary configurations, faulty leaders can always greatly reduce protocol throughput. Worse, this attack does not require any explicit equivocation; it suffices for a faulty leader to simply delay responding, making it hard to detect misbehavior –, and thus represents a significant exploit opportunity for a Byzantine attacker. To the best of our knowledge, this liveness concern is present in *all* existing CLSO protocols today. This paper asks: is this fundamental or can we do better?

Our solution. We find that yes, it is possible to improve the liveness guarantee offered by CLSO protocols. To this effect, we propose BeeGees, a new consensus protocol that strengthens liveness and instead satisfies the following stronger property we call AHL (any honest leader):

Definition 1.2. (AHL). After GST, if an honest leader in view v proposes a value and views $v < v_{i_1} < ... < v_{i_k}$ (non-contiguous) have honest leaders, then this value will necessarily be committed.

This property is not achievable in existing protocols. Intuitively, when QCs are not contiguous, it becomes possible for conflicting QCs to form unbeknownst to the current leader; these QCs can trigger safety violations when committing a block. BeeGees's core insight lies in observing that *Prepare* messages (called PREPARE messages in Hotstuff, PRE-PREPARE messages in PBFT), which are traditionally discarded by BFT protocols, can in fact be leveraged to side step this vulnerability. In the presence of omission faults or asynchrony, BeeGees uses these messages to *prevent* conflicting QCs from forming. In the presence of equivocation, BeeGees instead uses *Prepare* messages to reliably *detect* when a conflicting QC could have formed and eagerly excludes this block from commit consideration (abort). *Prepare* messages further allow BeeGees to detect *implicit QCs*, QCs for blocks proposed by honest leaders that would have formed but for a malicious leader failing to disseminate them. Together, these properties allow BeeGees to be the first CLSO protocol that satisfies AHL. Offering this stronger liveness property drastically curbs the impact of a Byzantine leader on the system: after GST, no node can delay the commitment of an honest leader's proposal by more than one view. Importantly, BeeGees achieves this without sacrificing performance: it has optimal latency of two phases [2], quadratic word complexity when used with threshold signatures [19], and linear word complexity with SNARKs [4], matching the state of the art ([16, 21]).

Understanding the limits of existing protocols. BeeGees is the first protocol to satisfy the stronger liveness property AHL while also maintaining safety. While BeeGees use of *Prepare* messages may appear like a simple fix, it actually points to something more fundamental. In particular, Prepare messages allow BeeGees to satisfy a property called sequentiality, which we prove is a *necessary* condition to satisfy AHL. Sequentiality, informally, requires that (after GST) for any pair of honest leaders L and L' in views v and $v' > v$ respectively, L' must extend the proposal of L. Put differently, all honest proposals must be on the same chain. No other existing CLSO protocol guarantees sequentiality, and thus can fundamentally not satisfy AHL.

The remaining paper is structured as follows. We first define the system model (§2) before introducing the relevant background and identifying liveness shortcomings (§3). We then present BeeGees, a novel BFT protocol that overcomes the outlined liveness concerns (§4). We empirically validate our claims in §5, and conclude in §6.

2 PRELIMINARIES

We adopt the standard BFT system model in which $n = 3f+1$ replicas communicate through a reliable, authenticated, point-to-point network where at most f replicas are faulty. A strong but static adversary can coordinate faulty replicas' actions but cannot break standard cryptographic primitives. We adopt the partially synchronous model, where there exists a known upper bound Δ on the communication delay, and an unknown Global Stabilization Time (GST) after all messages will arrive within Δ [15]. We assume the availability of standard digital signatures and a public-key infrastructure (PKI). We use $\langle m \rangle_r$ to denote a message m signed by replica r. A message is well-formed if all of its signatures are valid.

Byzantine fault-tolerant state machine replication (BFT SMR) is formally defined as follows:

Definition 2.1. (BFT SMR). A Byzantine fault tolerant state machine replication protocol commits client requests in a linearizable chain, which satisfies the following properties [1] [2].

- Safety. Honest replicas commit the same values at the same height (position in the chain).
- Liveness. All client requests eventually receive a response; all requests are eventually committed by every honest replica.
- External Validity. If an honest replica commits a value, v, then $\text{ExVal}(v) = true$, where $\text{ExVal}(v)$ is a predicate that checks whether v upholds all application invariants.

We also formalize the notion of Chained-Leader-Speaks-Once (CLSO) protocols (inspired from [3]).

Definition 2.2. (CLSO). A CLSO protocol is a BFT-SMR protocol that proceeds in a sequence of views and has two properties:

- Each view changes the leader, and there is an infinite number of views led by honest leaders.
- Block commitment cannot be guaranteed within a single view.

3 RELATED WORK & LIVENESS ISSUES IN CLSO PROTOCOLS

All existing CLSO protocols follow the same general pattern. While we focus on HotStuff here [34], our observations broadly apply to all others. Most such protocols follow a common logical structure [7, 8,

16, 21, 32, 34]. They proceed in a sequence of *views*, each led by a designated leader. The view leader proposes a batch of client operations (a *block*), and drives agreement to safely order and commit these operations. Blocks contain a parent pointer to their predecessor block, thereby forming a chain. The protocol is structured as follows:

Normal Case. The leader of view v begins by proposing a block B for log slot (henceforth *height*) i. Committing a proposal consists of two logical phases: a *non-equivocation* phase and a *durability* phase. The non-equivocation phase ensures that at most one block proposal will reach agreement *within* a view. The durability phase ensures that any (possibly) agreed-upon decision is preserved *across* views and leader changes, thus guaranteeing that only one block can be committed for height i. Each phase makes use of *quorum certificates* (QC) to achieve the desired invariants. A QC, written $QC = (B,v,\sigma)$, refers to a set of unique signed replica votes σ for block B proposed in view v. A QC describes a *threshold* $|\sigma|$ of confirmations that proves a super-majority of distinct replicas voted for block B in view v. Upon committing a block, honest replicas execute its transactions and enter the next view through a view change (described below).

View Change. The *view change* is responsible for changing leaders and preserving all decisions made durable by previous leaders. View change protocols are notoriously tricky: they can be expensive and hard to get right [5]. The primary challenge stems from reconciling different participants' beliefs about what *could have been committed*, as asynchrony and malicious leaders may cause replicas to consider different sets of blocks as potentially committed.

To understand the liveness challenges associated with CLSO protocols, we first describe in more detail how stable non-chained (*basic*) Hotstuff works (§3.1), before introducing the refinements of *chaining*/*pipelining* and *leader-speaks-once* (§3.2). We then demonstrate the resulting liveness pitfall (§3.3), and show that it is non-trivial to address (§3.4).

3.1 Basic HotStuff

Hotstuff proceeds in a sequence of *views* and consists of three voting rounds, one for the non-equivocation phase, and two for the persistence phase. In the *Prepare round*, the leader proposes a block A in view v for height i and each replica votes to prepare if it has not already prepared a block at height i with a higher view. If the leader successfully obtains a QC ($n-f$ distinct replica votes) in the Prepare round, a *prepareQC* forms, and the leader moves on to the Pre-Commit round. The existence of a prepareQC ensures agreement on B within the view: no other block could have been certified in view v as any two prepareQCs must overlap in at least one honest replica — a contradiction, as honest replicas will not vote twice.

In the Pre-Commit round, the leader broadcasts the prepareQC to all replicas via a *Pre-commit(B)* request. The replicas locally record the prepareQC with the highest observed view, and echo their acceptance of the Pre-commit(B). The leader waits to receive $n-f$ Pre-Commit replies, and assembles a *precommitQC*. We note that the use of a precommitQC is only necessary for Hotstuff to achieve linear view change complexity, and not for safety per se — we defer discussion to [34]).

In the final Commit round, the leader broadcasts the *precommitQC* to all replicas. Replicas become *locked* on this QC: they will never vote for a conflicting block unless they receive a prepareQC in a higher view. The existence of a higher conflicting prepareQC is evidence that the locked QC could not have committed (honest replicas would not vote to support two conflicting blocks). Finally, the leader forms a *commitQC* upon receiving $n-f$ Commit votes. It attaches the commitQC in a Decide round to inform replicas that the block committed, at which point they can execute the block's operations and move to the next view.

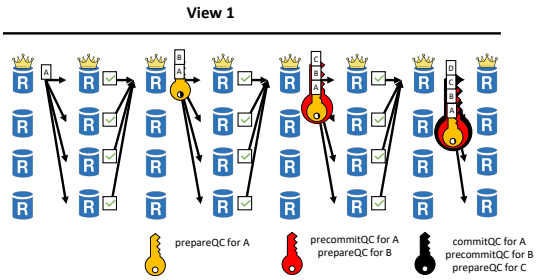

Figure 1: Chained Hotstuff

Recent BFT protocols manage to avoid the Pre-Commit round of Hotstuff (thus achieving optimal two-round commit) by, respectively, eschewing linear world complexity [16, 21], elongating view changes [29], or introducing novel cryptography [17].

3.2 Chaining and LSO

Chaining. The aforementioned protocols require $k \in \{2,3\}$ rounds to commit a block: each commitment thus requires forming k individual QCs. Prior work [16, 17, 21, 34] observes that, while each step serves a different purpose, all have identical structure: the leader proposes a block, collects votes and forms a QC. To amortize cryptographic costs and minimize latency, one may pipeline commands such that a single QC simultaneously serves as *prepareQC*, *precommitQC* (if applicable), and *commitQC* for consecutive block commitments.

Consider for example a scenario in which a chained protocol with $k=3$ (e.g. Hotstuff) is attempting to commit four blocks A, B, C, and D (Figure 1). The view leader first proposes A, collects $n-f$ votes for A, and forms its first QC (QC_A). QC_A functions as *prepareQC* for A. Next, the leader proposes B, and indicates that (i) A is the *parent block* of B, and (ii) that A has been certified by QC QC_A. It once again collects $n-f$ votes, forming QC_B. This QC acts as both the *precommitQC* for A and the *prepareQC* for B. Likewise, the leader proposes and obtains a QC_C for block C with parent $B.QC_C$ acts as a *commitQC* for A, *precommitQC* for B, and *prepareQC* for C. A is committed once k consecutive QCs attest to A. In the next round the leaderforwards QC_C (and proposes block D as extension to C). Upon receiving QC_C, all replicas learn that A has been committed and can thus safely execute the operations in the block.

LSO. In the previous example, a single leader drives the full protocol (a *stable leader*). It is responsible for deciding which block to include next in the chain, for collecting replica votes, for creating the corresponding QC and broadcasting it to all replicas.

Stable leader protocols [9, 10, 19, 24] only rotate leaders when there is a failure. This raises fairness concerns: malicious leader can

censor operations, penalize specific users or influence operation ordering [12, 20]. *Leader-Speak-Once* (LSO) protocols [16, 17, 21, 29], instead, try to minimize the influence of a leader on a new proposal. To do so, they bound the duration of each view (and thus the reign of a leader) to a single protocol phase. A leader receives votes, forms a QC, proposes the next block, and is immediately rotated out. Upon receiving the block proposal, replicas directly increment their view and send their votes to the *next* leader in the rotation. For instance, in the example of Figure 1, QC_A will be assembled by a new leader, which proposes B in view $v+1$. The commit rule remains unchanged: A commits as soon as k QC's in *consecutive* views attest to A.

Other approaches to fairness LSO mitigates concerns over long-tenured leaders in chained BFT protocols – but, like any leader based system, cannot fully side-step all possible order-fairness related concerns. There exists a plethora of recent work that explores (LSO-) alternative approaches in BFT systems: [11, 35] optimize leader-selection processes using reputation schemes, [22, 23, 36] leverage voting to democratize order in leader-based settings, and [13, 25, 28, 30] explore fully leaderless approaches.

3.3 Liveness Concerns

Existing Chained-Leader-Speaks-Once (CLSO) protocols commit a block once k consecutive QC's attest to the block's validity. Hotstuff requires $k=3$, while other CLSO protocols reduce the number of consecutive QCs to $k=2$ [21, 33]. Since CLSO rotates leaders, this implies that $k+1$ consecutive honest leaders are necessary to guarantee commitment: one honest leader to propose the block, and k consecutive leaders to assemble and forward QCs, and propose child blocks of their own. Only honest leaders guarantee a QC will be formed (no equivocation). Unfortunately, we find that the need for $k+1$ consecutive honest leaders to commit a block introduces a significant liveness vulnerability that can delay block commitment for long periods of time. We submit that all existing CLSO protocols suffer from this limitation. To illustrate this claim, consider the following Hotstuff run. Hotstuff requires $k+1=4$ consecutive honest leaders to commit. Consider replicas R_1, R_2, R_3 and R_4, with R_4 being faulty (Figure 3). Leaders are elected round-robin. R_1 proposes block A, yet R_4 might never assemble and broadcast the final QC (acting as *commitQC*) necessary for replicas to execute A. Likewise, if R_2 and R_3 propose blocks B and C respectively, R_4 may fail to form the corresponding *precommitQC* (for B) and *prepareQC*(for C). In fact, with $n=4$ replicas, a single faulty leader can prevent Hotstuff from committing *any* block! More generally, requiring $k+1$ consecutive honest leaders can create significant latency spikes, even for large participant sets, as blocks must be re-proposed until they find a sequence of n consecutive leaders. We measure this danger through simulation in Figure 2, where we calculate the number of phases necessary to commit a block given a random assignment of faulty nodes and random leader election policy. In our experiment, Hotstuff requires an average of 12 phases to commit (a three-fold increase over the failure-free case), and has an observed worst-case latency of 129 rounds. We note that, in the absolute worst case (e.g. an unfavorable round-robin schedule), Hotstuff (or any CLSO protocol with $k=3$) may *never* commit a block.

3.4 It's Not Easy to Relax

So far, we have illustrated that existing CLSO protocols' safety rules – namely, requiring k consecutive QCs to commit – can expose a significant liveness vulnerability. This begs the question: can we can *relax* the commit requirement in order to strengthen liveness? The answer is unfortunately no. There is no trivial way to do so without simultaneously compromising safety; past attempts to do so resulted in safety violations [5]. To illustrate, consider the following hypothetical run. Note that for simplicity, we will adopt the more efficient "two-QC" rule of Jolteon [16] or Fast Hotstuff [21], i.e. $k=2$, although the same reasoning holds for Hotstuff's three-consecutive QC rule ($k=3$). As reminder, we say a $QC := (B', v, \sigma)$ ceritifies block B if $B == B'$ or B' extends B.

Let leaders be elected round-robin where R_i is leader for view v where $v \pmod i == 0$. We require that – akin to Hotstuff (§3.1) – replicas vote to certify a block as long as they are not locked on a higher conflicting block. This is true for all existing CLSO protocols with $k=2$. We write B_i to indicate block B is proposed for height i. Now let us assume that any two (possibly non-contiguous) QCs certifying block B would suffice to commit B. We show with a simple counterexample that temporary periods of asynchrony can cause conflicting blocks to commit, violating safety. Consider the following sequence of views:

- *Views 1-2.* R_1 proposes A_1. A QC forms for A_1 at R_2. R_2's broadcast messages are delayed. Asynchrony leads to a view change.
- *View 3-5.* R_3 receives responses from all replicas except from R_2. All responses are empty (recall that replicas only include QCs in view changes, not votes). R_3 proposes B_1. A QC forms for B_1 at R_4. R_4's broadcast messages are delayed. Asynchrony leads to view change.
- *View 6-8.* R_1 receives responses from all replicas expect for R_4 and learns about QC_{A_1}. It proposes A_2, which extends A_1. A QC forms at R_2. R_2 sees that two QCs certify A_1 and thus commits A_1. Asynchrony leads to a view change.
- *View 9-11.* R_3 receives responses from all except R_2 and learns about QC_{B_1}. It proposes B_2, which extends B_1. A QC forms at R_4. R_4 sees that two QCs certify B_1 and thus commits B_1.

R_2 and R_4 commit conflicting blocks (A_1 and B_1 respectively), violating safety. The root cause here is simple: committing a proposal after observing QCs in non-contiguous views is dangerous because there may exist a higher conflicting QC that formed in between (e.g. a QC for B_2 formed with a greater view than the QC for A_2). In contrast, requiring QCs to be in contiguous views ensures that, for any committing proposal p, a QC that extends p will be preserved across view changes: since at least two QC's are required to commit p (three for HotStuff), existence of the latest QC_{lat} implies that at least $f+1$ honest replicas have observed (at least) the preceding QC_{pre}. Since QC_{pre} (by assumption of being contiguous) has the highest view (bar Q_{lat}), it follows that every view change (a quorum of $n-f$) must observe QC_{pre} at least once. Thus, all future proposals must extend p.

This paper thus asks: can we strengthen liveness to commit with non-consecutive honest leaders (AHL) without violating safety? We answer in the affirmative. We introduce BeeGees, CLSO protocol that, after GST, will commit all blocks proposed by an honest

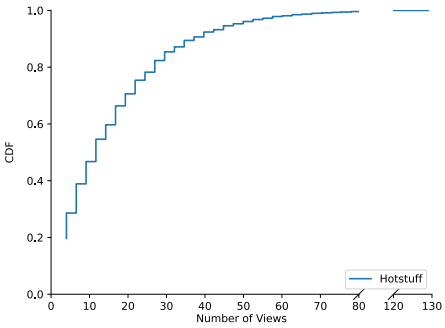

Figure 2: # number of views needed for commit

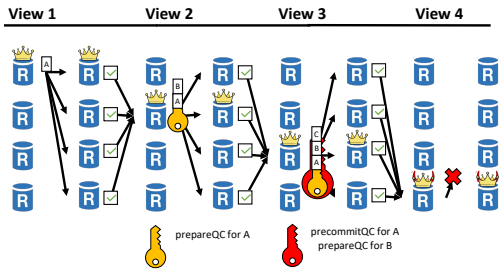

Figure 3: Liveness issue with CLSO protocols

leader in view v after at most two more honest-led (possibly non-contiguous) views v' and v''.

4 BEEGEES

BeeGees achieves optimal phase complexity (two phases to commit a block), and can be engineered to have quadratic word complexity with threshold signatures, or linear word complexity with SNARKs. BeeGees is responsive with consecutive honest leaders and satisfies the following property.

Theorem 4.1. *(AHL). After GST, if an honest leader in view v proposes a value and views $v < v_{i_1} < \ldots < v_{i_k}$ (non-contiguous views) have honest leaders, then it is guaranteed to commit the value.*

This property has consequences for both safety and liveness. For safety, BeeGees must ensure that, in the presence of an honest leader L that proposes B, the existence of two QCs in non-contiguous views for B be sufficient to guarantee that no conflicting block B' can commit. This theorem also places stringent liveness requirements on BeeGees: after GST, all blocks proposed by an honest leader *must* be committed. Note that this is not a property that is traditionally guaranteed by CLSO protocols. In the rest of this section, we first describe the core intuition behind BeeGees before describing the protocol in more detail.

4.1 BeeGees Overview

The case of the conflicting QC. As shown in §3.4, committing a block requires ensuring that no higher conflicting QC can exist. Committing with consecutive QCs guarantees precisely that (§3.4). To satisfy AHL, a protocol must instead commit blocks even when the QCs certifying them are not consecutive. We thus require alternative mechanisms to prevent conflicting QCs from forming. In an ideal world, one would design a clever algorithm that prevents *all* such QCs from forming. Unfortunately, we cannot ensure this guarantee when there is equivocation. Instead, BeeGees proceeds in two ways: under asynchrony and omission faults, BeeGees does indeed *preclude* all conflicting QCs. In the presence of equivocation, BeeGees instead reliably *detects* when a conflicting QC could have formed and immediately excludes this block from commit consideration (abort). Together, these mechanisms ensure that, after GST, all blocks proposed by honest leaders will eventually commit.

Key Idea. BeeGees's key insight is simple. It explicitly makes use of information that all other CLSO protocols (and most other BFT protocols) traditionally discard after processing. *Prepare* messages (*Pre-prepare* messages in PBFT)[2]. These messages have until now only been used to achieve an optimistic fast path ([19, 24, 30]) in which a superquorum containing all n replicas informs the client that the block has been persisted. In non-fast path protocols, these messages were thought to convey no useful information as they precede the protocol's non-equivocation phase, and thus hold no bearing to maintaining *safety*. BeeGees instead uses them to improve *liveness* by reliably distinguishing between asynchrony/omission faults and equivocation.

Technical Intuition. By quorum intersection, if a QC forms for a block B, all subsequent view change leaders will receive at least one *Prepare* message for B or a block that extends B. By extending this block, subsequent leaders will never propose conflicting blocks. If two conflicting QCs do form as a result of equivocation, subsequent leaders will necessarily observe the existence of two *Prepare* messages in the same view, and thus abort block commitment. These conflicting messages further create explicit evidence of misbehavior, allowing the offending faulty leader to be removed.

While the above approach elegantly guarantees safety without requiring consecutive QCs, it does not yet fully satisfy *AHL* (Theorem 4.1). For example, consider the following scenario with a sequence of five leaders: L_1 (honest), L_2 (faulty), L_3 (honest), L_4 (faulty), and L_5 (honest). L_1 could propose a block B (after GST), and all honest replicas vote for it, implicitly forming a QC for B. L_2 could, however, fail to assemble and disseminate this QC. L_3 would not observe a QC for B and instead propose a new block B' that extends B. Similarly, L_4 would fail to generate and disseminate a QC, and L_5 would fail to observe a QC for B'. To satisfy Theorem 4.1 we must commit B since there were three honest leaders; however, in this scenario we fail to do so. To address this issue, BeeGees develops a novel technique, QC materialization, that makes these implicit QCs explicit, allowing replicas to commit the relevant blocks. QC materialization hinges on two observations: after GST, if an honest leader broadcasts a *Prepare* message, all honest replicas will receive it and send a reply. Second, an honest leader is guaranteed to receive replies from all honest replicas in Δ time (after GST).

[2] to avoid naming conflicts, we will refer to these messages as *Prepare* messages

Protocol Structure. BeeGees shares the same structure as other CLSO protocols. It consists of four components: a fast view change, a slow view change, a commit procedure, and a view synchronizer. Fast view changes occur in the absence of delayed messages or failures. Slow view changes are triggered by lack of progress (view timeouts). For each view, the leader runs a commit procedure to determine which blocks in the chain can safely be committed. The view synchronizer ensures all honest replicas remain in the same view for sufficient amount of time. BeeGees is compatible with any view synchronizer ([6, 26]); we focus on other components here.

4.2 BeeGees Data Structures

Blocks and Block Format. As is standard, BeeGees batches client requests into *blocks*, with each block containing a hash pointer to its parent block (or to null in the case of the *genesis* block). A block's position in the chain is known as its height. A block $B := \langle v, p, QC_{anc}, b, NV_{set} \rangle$ contains the following information: v, the view the block was proposed in; p, its parent block; QC_{anc}, the quorum certificate certifying an ancestor block of B; and b, a batch of client transactions (Alg. 1 lines 2-5). Additionally, blocks proposed in the slow view change must contain NV_{set} (Alg. 1 line 6), the set of New-view messages (more detail later). A block B is valid if 1) its parent block is valid (or B is genesis), 2) all included client transactions b satisfy all application level validity predicates, and 3) all included signatures are valid. Honest replicas only accept valid blocks – we omit validation checks from the pseudocode.

Block extension and conflicts. Parent pointers link blocks into a chain. We define a block ancestor of B to be any block B_{anc} for which a path (of parent links) exists from B to B_{anc}. We say a block B' extends (or is descendant of) a block B ($B \longleftarrow B'$) if B is an ancestor of B'. We say that B' conflicts with B if neither extends the other ($\neg(B \longleftarrow B' \vee B' \longleftarrow B)$). Informally, if B' conflicts with B, these blocks are on separate forks of the chain and only one of them can commit. By convention, we say that blocks extend themselves.

Equivocation. Honest leaders may propose only a single block per view. We label conflicting blocks with the same view as *equivocating*. An equivocation proof (more details later) constitutes evidence of leader equivocation.

Message Types. In BeeGees there are three types of messages: Vote-req, Vote-resp, and New-view. ⟨Vote-req, B⟩ messages are the *Proposal* messages in BeeGees and contain B, the leader's proposed block. Replicas send ⟨Vote-resp, B⟩ messages to vote on a proposal for B. Since blocks are chained together, a Vote-resp message for a block counts also as Vote-resp for all of its ancestors. Each replica stores its current view, v_r, the latest accepted (highest view) Vote-req, vr_r, and Vote-resp, vp_r (Alg 2. lines 1-3). ⟨New-view, v, vr_r, vp_r⟩ messages are used by the slow view change to maintain progress despite failures or asynchrony. They contain the view v that the replica is advancing to, the replica's latest Vote-req, vr_r, and its latest Vote-resp vp_r.

Quorum Certificates. A $QC := \langle Q, B \rangle$ consists of a set $Q := n-f$ Vote-resp messages and a *certified* block B. We say a block B is *certified* if there exists a quorum Q of $n-f$ Vote-resp messages for B itself (direct) or a descendant block B' (indirect). Given a set of any $n-f$ Vote-resp messages, we can thus determine which block was certified by identifying the highest (w.r.t view) common ancestor (Alg. 1 line 11). A QC contains Q and $B := B_{anc}$, the highest block that Q certifies. (Alg. 1 lines 12-13). We say that two QCs conflict if they certify blocks that conflict. In the rest of the paper, we, denote a QC as *implicit* as soon as the necessary Vote-resps to form Q are cast, but the QC has not yet been assembled. A leader materializes an implicit QC into an explicit QC by assembling the necessary votes.

Ranking. We introduce a notion of ranking rules for both blocks and QCs. Blocks with higher views have higher ranks; ties are broken by the rank of their contained QCs (Alg. 1 line 9). These rules are used to determine whether a block is safe to accept in the slow view change.

Algorithm 1 Data Structure Utilities

```
1:  procedure CreateBlock(v, B_p, QC, NV_set)
2:      B.v ← v                          ▷ The view the block B is proposed in
3:      B.p ← B_p                        ▷ The parent block B extends
4:      B.QC_anc ← QC                    ▷ The QC for an ancestor block that B contains
5:      B.b ← a batch of client transactions
6:      B.NV_set ← NV_set                ▷ The set of NewView messages for slow blocks
7:      return B
8:  procedure BlockRank(B_1, B_2)        ▷ Blocks are ranked by view, ties are broken by the higher QC
9:      return B_1.v ≥ B_2.v ∧ QCRank(B_1.QC_anc, B_2.QC_anc)
10: procedure CreateQC(Q)                ▷ Q := n−f Vote-resp messages
11:     B_anc ← highest common ancestor block of Q
12:     QC.B ← B_anc                     ▷ The block the QC certifies
13:     QC.Q ← Q
14:     return QC
15: procedure QCRank(QC_1, QC_2)         ▷ QCs are ranked by view of the block they certify
16:     return QC_1.B.v > QC_2.B.v
```

4.3 Protocol Details

Fast View Change (FVC). We first focus on the steady state. Successive leaders transmit state through a fast view change. The structure of BeeGees's is identical to existing CLSO protocols in that there are two steps: 1) the leader proposes a valid block to all replicas (sending step) and 2) replicas accept the block and forward their vote to the leader of the next view (receiving step).

*Sending step.*The leader of view $v_r + 1$ forms a valid QC for a block B in view v_r when it receives receives $n-f$ matching votes for B (Alg. 2 lines 5-6). The leader can then safely propose a new block B', which has B as its parent block (Alg. 2 lines 7-8).

Receiving step. Replicas deem a proposal for B' valid if the associated QC is for v_r (i.e. contiguous), and B' extends B (Alg. 2 line 14). It updates its current view $v_r := v_r + 1$, its latest received $vr_r :=$ ⟨Vote-req, B'⟩ and its latest sent $vp_r :=$ ⟨Vote-resp, B'⟩, indicating its support for block B' (Alg. 2 lines 15-17). It then sends vp_r to the leader of the next view ($v_r + 1$) (Alg. 2 line 18).

The FVC in BeeGees is simple: as views are contiguous, the new leader is guaranteed to see the latest possible QC. It can then easily

Algorithm 2 Fast View Change (Steady State)

1: $v_r \leftarrow 1$ ▷ Current view of replica r
2: $vr_r \leftarrow \perp$ ▷ Latest Vote-req received
3: $vp_r \leftarrow \perp$ ▷ Latest Vote-resp sent
4: // *only the leader of view* v_r+1
5: **upon** receiving $S \leftarrow 2f+1$ *matching* ⟨Vote-resp, B⟩ messages while in view v_r **do**
6: $QC_B \leftarrow$ CreateQC(S) ▷ Certify B since it has $2f+1$ votes
7: $B' \leftarrow$ CreateBlock($v_r+1, B, QC_B, \perp$) ▷ Propose B' which has B as its parent
8: **send** ⟨Vote-req, B'⟩ to all
9:
10: **upon** receiving a valid ⟨Vote-req, B'⟩ from L_{v_r+1} **do**
11: $QC_B \leftarrow B'.QC_{anc}$
12: $B \leftarrow QC_B.B$ ▷ Gets the certified block that B' extends
13: // *in the normal case the QC will be from the previous view*
14: **if** $B'.NV_{set} = \perp \wedge B.v+1 = B'.v \wedge B = B'.p$ **then**
15: $v_r \leftarrow B'.v$ ▷ Move to the next view
16: $vr_r \leftarrow$ ⟨Vote-req, B'⟩ ▷ Update latest Vote-req
17: $vp_r \leftarrow$ ⟨Vote-resp, B'⟩ ▷ Update latest Vote-resp
18: **send** vp_r to L_{v_r+1} ▷ Send vote to the next leader
19:

extend the chain without any risking of a conflicting QC forming. There is no such continuity in the slow view change (SVC), which requires more care.

Slow View Change (SVC). The SVC has two main objectives: 1) maintain consistency *across* views and 2) continue making progress on honest proposed blocks.

Local State. Each replica maintains a view timer VT that resets every time it advances to a new view. This timer is used to detect a lack of progress in a view. For reliable progress, BeeGees must ensure that (after GST) all honest replicas enter a common view within a bounded period, and remain in it long enough to form a QC. This problem is orthogonal to agreement, and can be delegated to an external *View Synchronizer* component—in the remainder of this section we assume such synchronization as given, and defer to §4.4 for details.

The leader of the new view additionally maintains 1) NV_{set}, the set of New-view messages received, 2) B_{parent}, the parent block of the new leader's next proposal, 3) QC_{anc}, the highest ranked explicit QC that B extends, and 4) a materialization timer MAT (more detail follows).

Trigger Conditions. A slow view change is triggered when enough replicas fail to make progress in the current view (when their view timer VT expires). A replica initiates a view change by invoking wish_to_advance (Alg.3 line 8). It enters view $v' > v_r$ upon receiving a propose_view(v') signal. Upon entering v' a replica sends a New-view message containing all its local state (Alg. 3 lines 10-12 to the leader of view v'. When the leader receives a ⟨New-view, v_r,vr_r,vp_r⟩ message, it adds it to the set of New-view messages received so far for view v_r. A slow view change is triggered when sufficiently many ($n-f$) New-view messages have been received (Alg. 3 line 18). Before proposing a new block, the leader must identify a parent block for its proposal to extend. The key challenge is ensuring that the parent block it selects preserves safety and liveness.

Parent Block Selection. The new leader first selects a parent block B_{parent} to extend. Recall that in BeeGees, unlike in other CLSO protocols, New-view messages include a replica's last seen Vote-req message. The leader then always selects the highest ranked block among these Vote-req messages (Alg. 3 line 19). In doing so, the leader guarantees that it always extends the latest block for which a QC *could* have formed (but that the leader did not necessarily receive). By the quorum intersection property, if forming a QC requires at least $n-f$ replicas receiving the corresponding Vote-req messages, at least one of these messages would have been included in the $n-f$ New-view messages. In the absence of explicit equivocation, using Vote-req messages in this way precludes the leader from extending a block that conflicts with a QC in a higher view. If a previous leader does equivocate, there may exist Vote-req messages for equivocating blocks that have the same (highest) rank.

We remark that – in order to prevent the formation of a conflicting QC – the leader must not propose a block that conflicts with a potentially certified block. In this case, however, the leader does not have enough information to know which, if any, of the equivocating blocks has been certified, and thus cannot proceed. To nonetheless make progress, BeeGees temporarily relaxes the proposal rule: BeeGees allows the leader to arbitrarily select and extend one of the equivocating blocks, but later aborts commitment if an unlucky choice results in the formation of a conflicting QC. We discuss in detail how BeeGees safely resolves this scenario in the commit rule.

Implicit QC Materialization. Next, the new view leader must ensure that, after GST, any block proposed by an honest leader will eventually be committed. In order to do so, it must ensure that, if the leader was honest and we are after GST, a QC must eventually form. Otherwise, it may take more than $k+1$ honest leaders to commit this block, violating AHL. BeeGees leverages Vote-req messages to enforce this invariant through a novel QC materialization technique. BeeGees makes three observations: 1) after GST, all honest replicas are guaranteed to vote in favor of an honest leader's proposal. 2) after GST, the next leader is guaranteed to receive responses from all honest nodes within a materialization timeout (MAT) 3) before GST, AHL does not require that blocks proposed by honest leaders be committed. It follows that, after GST, if an honest leader proposed a block B, an implicit QC formed and subsequent leaders will necessarily receive $n-f$ ⟨Vote-resp, B⟩ (or descendants of B). As such, any time a leader sees $n-f$ ⟨Vote-resp, B'⟩ messages for some block B', such that $B \longleftarrow B'$, it could have been proposed by an honest leader and must therefore be certified. Note that BeeGees enforces this guarantee for liveness, not safety. Before GST, honest leaders' proposals may - as is the case in existing CLSO protocols – fail to generate a QC.

We now describe the precise materialization protocol. Recall that B_{parent} is the highest ranked block among the Vote-req messages in NV_{set}, and the block that the leader must extend. The leader identifies the highest ranked QC_{anc} (Alg. 3 line 20) on the chain that certifies an ancestor of B_{parent}, B_{anc}, just as one would in traditional CLSO protocols. Note that there can be many intermediate blocks between B_{anc} and B_{parent} for which no QC formed. Next, the leader tries to materialize any higher ranked implicit QCs on

the chain. If there are enough Vote-resp messages to materialize a QC for B_{parent} (Alg. 3 lines 26-27), the leader materializes this QC. Since B_{parent} is the highest ranking block on the chain, the leader immediately proposes a new block that extends B_{parent}. This is safe as B_{parent} is, by construction, necessarily the highest ranked block on the chain; no conflicting higher-ranked QC can exist. If, instead, there are insufficient Vote-resp messages, the leader starts a materialization timer (MAT) during which it waits for additional New-view messages in order to materialize implicit QCs for descendants of $QC_{anc}.B$. If the leader eventually receives $n-f$ ⟨Vote-resp, B_{desc}⟩ where B_{desc} is a descendant of $QC_{anc}.B$, B_{desc} could have been proposed by an honest node. The leader thus materializes an explicit QC for B_{desc} and updates its local knowledge of the highest ranked known QC_{anc} (Alg. 3 lines 23-24). If, while waiting, the leader receives $n-f$ ⟨Vote-resp, B_{parent}⟩, the leader instead updates QC_{anc} to certify B_{parent}, and the materialization timer can be canceled. Finally, the leader proposes a new block B with parent block $B.p := B_{parent}$, quorum certificate $B.QC_{anc} := QC_{anc}$, and $B.NV_{set} := NV_{set}[v]$ the set of New-view messages received (Alg. 3 lines 28-29, 32-33).

View Change Validation. When a replica receives a valid ⟨vote-req, B'⟩ proposal from the leader, the replica checks that the leader did in fact perform the view change correctly (Alg. 3 line 39). It confirms that 1) the leader obtained $n-f$ New-view messages 2) that the proposed block's parent was in fact the highest ranked blocks among Vote-req messages 3) that the proposal extends the highest QC received by the leader. When confirmed, the replica updates its state (Alg. 3 lines 40-42), and sends a Vote-resp to the next leader (Alg. 3 line 43).

Commit Rule. The commit rule determines which blocks in the chain can be safely marked as committed; it is invoked each time a replica receives a valid ⟨Vote-req, B'⟩ message from the leader. The test considers the last two QCs and their associated blocks (Alg. 4 lines 2-5). Informally, a block is safe to commit when no possible conflicting block can also be committed, in other words when no conflicting QC could have formed. More specifically, the commit test considers two cases. We write QC_{child} and QC_{parent} to denote respectively the last and second to last QCs in the chain.

Consecutive QCs. If the blocks certified by QC_{parent} (B_{parent}) and QC_{child} (B_{child}) were proposed in consecutive views (Alg. 4 line 6), it safe to commit B_{parent}. As shown in §3.4, no higher ranked (than B_{parent}) conflicting QC will form.

Non-consecutive QCs. The use of Prepare messages precludes conflicting QCs from forming in the presence of omission faults or asynchrony. It does not, however, prevent conflicting QCs from forming when the leader equivocates. Thus, the first step is to identify whether a conflicting QC could have formed as a result of equivocation. This is done by iterating through all of the ancestor blocks in between B_{parent} and B_{child} and looking for evidence of equivocation for a conflicting block (Alg. 4 lines 10-12). As mentioned in §4.1, equivocating blocks are different blocks proposed in the same view. Thus, evidence of equivocation (equivocation proof) consists of Vote-req proposal messages for equivocating blocks in the same view. It is important that this equivocation proof contains a Vote-req for a *conflicting* block. Otherwise, this equivocation proof indicates that a non-conflicting QC could have formed, which

Algorithm 3 Slow View Change

1: $NV_{set} \leftarrow \{\}$ ▷ Stores New-view messages
2: $B_{parent} \leftarrow \perp$ ▷ Highest ranked Vote-req block
3: $QC_{anc} \leftarrow \perp$ ▷ The highest ranked explicit QC
4: $B_{anc} \leftarrow \perp$ ▷ The block QC_{anc} certifies
5:
6: **upon** view timer (VT) for v_r expiring **do**
7: *//call into external Synchronizer-Module*
8: Synchronizer.wish_to_advance()
9:
10: **upon** Synchronizer.propose_view(v'), where $v' > v_r$ **do**
11: $v_r \leftarrow v'$
12: **send** ⟨New-view, v_r,vr_r,vp_r⟩ to L_{v_r}
13: **if** $r == L_{v_r}$ **then start** materialization timer (MAT)
14:
15: *// only the leader of view v_r*
16: **upon** receiving ⟨New-view, v_r,vr_r,vp_r⟩ for view v_r **do**
17: $NV_{set}[v_r] \leftarrow NV_{set}[v_r] \cup$ ⟨New-view, v_r,vr_r,vp_r⟩
18: **if** $|NV_{set}[v_r]| = n-f$ **then** ▷ Trigger Slow VC
19: $B_{parent} \leftarrow$ HighVoteReq(S) ▷ Finds the highest ranked block to extend
20: $QC_{anc} \leftarrow B_{parent}.QC_{anc}$ ▷ Gets the QC contained within B_{parent}
21: $B_{anc} \leftarrow QC_{anc}.B$
22: *// while waiting for materialization timer (MAT), continually check to see if QC_{anc} can be updated*
23: **if** $NV_{set}[v]$ *contains* $n-f$ ⟨Vote-resp, B_{desc}⟩ where B_{desc} **extends** B_{anc} **then**
24: $QC_{anc} \leftarrow$ CreateQC($NV_{set}[v]$) ▷ Update QC_{anc} to be newly formed QC
25: $B_{anc} \leftarrow QC_{anc}.B$
26: **if** $B_{anc} = B_{parent}$ **then** ▷ QC_{anc} has highest possible rank, propose a new block
27: **cancel** materialization timer (MAT)
28: $B' \leftarrow$ CreateBlock(v,B_{parent},QC_{anc},$NV_{set}[v]$)
29: **send** ⟨Vote-req, B'⟩ to all ▷ Propose B'
30:
31: **upon** materialization timer (MAT) expiring **do**
32: $B' \leftarrow$ CreateBlock(v,B_{parent},QC_{anc},$NV_{set}[v]$)
33: **send** ⟨Vote-req, B'⟩ to all ▷ Propose B', with parent B
34:
35: **upon** receiving a valid ⟨Vote-req, B'⟩ from L_{v_r+1} **do**
36: $QC_B \leftarrow B'.QC_{anc}$
37: $B \leftarrow QC_B.B$
38: *// verify that L_{v_r+1} did the view change correctly*
39: **if** $|B'.NV_{set}| \geq 2f+1 \wedge$ HighVoteReq($B'.NV_{set}$) $= B'.p \wedge$ B' *extends* B **then**
40: $v_r \leftarrow B'.v$ ▷ Move to the next view
41: $vr_r \leftarrow$ ⟨Vote-req, B'⟩ ▷ Update latest Vote-req
42: $vp_r \leftarrow$ ⟨Vote-resp, B'⟩ ▷ Update latest Vote-resp
43: **send** vp_r to L_{v_r+1} ▷ Send vote to the next leader
44:
45: **procedure** HighVoteReq(NV_{set})
46: $B_{high} \leftarrow \perp$
47: **for** $s \in S$ **do** ▷ Iterate through all Vote-reqs in the VC
48: **parse** s as ⟨New-view, v_r,vr_r,vp_r⟩
49: **parse** vr_r as ⟨Vote-req, B⟩
50: **if** BlockRank(B,B_{high}) **then**
51: $B_{high} \leftarrow B$ ▷ Update the highest ranked block
52: **return** B_{high} ▷ Return highest ranked block in the VC

does not violate safety. Upon detecting equivocation, replicas must explicitly abort committing B_{parent} (Alg. 4 lines 13-14). Note that aborting in this case does not violate Theorem 4.1: we show in our proofs that the existence of an equivocation proof for a conflicting block guarantees that the leader who proposed B_{parent} must have equivocated, and thus is Byzantine faulty. Otherwise, if no equivocation proof is found, the replica can safely commit B_{parent} (Alg. 4 line 15). Our full correctness proofs can be found in [18, Appendix C].

Algorithm 4 Commit Rule

```
1:  upon receiving a valid vr_r ← ⟨Vote-req, B⟩ do
2:      QC_child ← B.QC_anc                          ▷ last QC in chain
3:      B_child ← QC_child.B
4:      QC_parent ← B_child.QC_anc                   ▷ second to last QC in chain
5:      B_parent ← QC_parent.B
6:      if AreConsecutiveQCs(QC_parent, QC_child) then
7:          commit B_parent
8:      else
9:          C ← ∅
10:         for B_anc ∈ GetAncestors(B_parent, B_child) do
11:             // look for possible conflicting QCs
12:             C ← FindEquivProof(B_parent, B_anc.p, B_anc.NV_set)
13:             if C ≠ ∅ then
14:                 return                           ▷ Not safe to commit
15:         commit B_parent
16:
17: procedure AreConsecutiveQCs(QC_parent, QC_child)
18:     return QC_parent.B.v+1 = QC_child.B.v
19: procedure GetAncestors(B_parent, B_child)
20:     A ← ∅
21:     B_anc ← B_child
22:     while B_anc.v > B_parent.v do
23:         A ← A ∪ B_anc
24:         B_anc = B_anc.p
25:     return A
26: procedure FindEquivProof(B_parent, B_target, S)
27:     C ← ∅                                        ▷ Keep track of conflicting blocks
28:     for s ∈ S do                                 ▷ Iterate through all blocks
29:         parse s as ⟨Vote-req, B'⟩
30:         // Different vote-reqs with the same view
31:         if B'.v = B_target.v ∧ B' ≠ B_target ∧ B' conflicts with
    B_parent then
32:             C ← C ∪ B'
33:     return C                                     ▷ Otherwise, no equivocation
```

4.4 View Synchronization

To ensure the reliable formation of a QC, BeeGees must ensure that all honest replicas enter – and remain in – a common view v for a sufficiently long period. Solving this problem is orthogonal to most BFT protocols, including BeeGees, and can be delegated to a blackbox *View Synchronizer* sub-module. To the best of our knowledge, BeeGees is compatible with most (if not all) existing view synchronizers [6, 16, 26]. For the purpose of this discussion, we adopt the Cogsworth API [26], which exposes two methods: honest

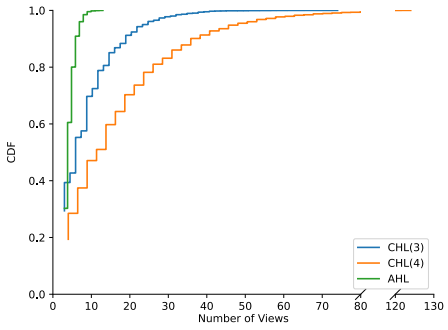

Figure 4: CDF of the number of views needed to commit an operation n = 100.

replicas invoke WISH_TO_ADVANCE to request a view change, and receive PROPOSE_VIEW(v') to confirm they may advance to view v'.

We define two parameters: T_ℓ and T_r, whose values are dependent on the specific view synchronizer that is used. T_ℓ is the maximum amount of time (after GST) between when the first honest replica enters a view v, and when the leader of view v, L_v, enters view v. T_r is the maximum amount of time (after GST) between when the leader of a view v, L_v, enters view v, and when *all* honest replicas enter view v. In total, the maximum amount of time between when the first honest replica enters a view v, and when all honest replicas enter view v is $T_\ell + T_r$. A common instantiation is for $T_\ell = T_r = \Delta$, thereby $T_\ell + T_r = 2\Delta$. Accordingly, BeeGees sets its view timer delay to $VT := T_\ell + T_r + 4\Delta$ to ensure that all honest replicas spend enough time in a view for a QC to form. Similarly, BeeGees sets its materialization timer to $MAT \geq T_r + \Delta$ [16] to guarantee that (after GST and view synchronization) the leader receives messages from all honest nodes, and thus succeeds in materializing implicit QCs.

4.5 Reflecting on AHL

BeeGees is the first Chained-Leader-Speaks-Once (CLSO) protocol to safely satisfy AHL, i.e. commit honest leader's proposals as soon as *any* other k honest leaders exist in some higher views (after GST). To better understand why existing CLSO protocols fall short, and more easily analyze future designs, we identify a *necessary* condition for satisfying AHL. In particular, we adopt as condition a partial synchrony equivalent of *sequentiality* [3]:

Definition 4.2. (Sequentiality). Let L_i and L_j be a pair of honest leaders, and wlog $i < j$. After GST, if L_i sends a proposal B_i, then $L_j's$ proposal B_j must extend B_i.

Intuitively, sequentially states that, after GST, all proposals issued by honest leaders *must extend each other*. Clearly, AHL relies on sequentiality to be true. Otherwise, two honest leader's proposals can extend divergent chains, which breaks safety as eventually both conflicting chains will be committed. We prove in [18, Appendix B.2] that:

THEOREM 4.3. *AHL is achievable only if* ***sequentiality*** *is satisfied.*

BeeGees guarantees sequentiality via the use of Prepare messages and implicit QC materialization, ensuring that if a honest

Table 1: Comparison of CLSO BFT protocols (excluding view synchronizer)

Protocol	Complexity (thresh)	Complexity (SNARKs)	# of phases	Responsive (consec.)	AHL
Casper FFG [8]	$O(n)$	$O(n)$	2	No	No
HotStuff [34]	$O(n)$	$O(n)$	3	Yes	No
Fast-HotStuff [21]	$O(n^2)$	$O(n)$	2	Yes	No
Jolteon [16]	$O(n^2)$	$O(n)$	2	Yes	No
BeeGees	$\mathbf{O(n^2)}$	$\mathbf{O(n)}$	**2**	**Yes**	**Yes**

leader issued a proposal B (after GST), every future leader **must** extend B. We prove in [18, Appendix D.3] that:

Theorem 4.4. *BeeGees satisfies **sequentiality**.*

In contrast, other CLSO protocols only consider QCs rather than Prepare messages in their slow view change. A Byzantine leader can thus intentionally fail to certify an honest block proposed after GST, by just crashing. This will preclude this block from appearing in future view changes as it was not certified. We prove in [18, Appendix B.1] that:

Theorem 4.5. *Prior partially synchronous CLSO protocols do not satisfy **sequentiality**.*

5 PERFORMANCE ANALYSIS

BeeGees is the first CLSO protocol to satisfy AHL. We next quantify the theoretical and practical benefits of our approach.

Theoretical Properties We first summarize the main theoretical properties of BeeGees as compared to the state of the art CLSO protocols in Table 1. Specifically, we measure the word communication complexity of each protocol excluding the view synchronizer, where a word contains a constant amount of signatures or bits. Word complexity measures the amount of words sent by honest parties over all possible executions and adversarial strategies. We say a protocol is responsive if after GST the latency between consecutive honest leaders is $O(\delta)$, where δ is the actual network delay. BeeGees satisfies AHL while maintaining quadratic word complexity with threshold signatures, linear word complexity with SNARKs, optimal phase complexity, and responsiveness with consecutive honest leaders. The formal analysis can be found in [18, Appendix E].

Performance Simulation Next, we formally quantify the performance gains made possible by strengthening the liveness condition from CHL to AHL. BeeGees will commit blocks in the presence of any $k+1$ honest leaders after GST and no longer requires $k+1$ consecutive leaders. In Figure 4 we compare BeeGees to 1) two-phase CLSO protocols (DiemBFTv4 [32], Fast-Hotstuff [21], Jolteon [16]), 2) three-phase CLSO protocols (Hotstuff [34]). We calculate the expected number of rounds necessary to commit an operation under AHL and CHL; leaders are choosen leaders at random. We additionally simulate commit latency when electing leaders in a round-robin fashion.

Theorem 5.1. *With a random leader election scheme, after GST, the expected number of rounds necessary to commit a block under the CHL is $L = \frac{(1-p^k)}{(1-p)p^k}$ [14] where $p = \frac{n-f}{n}$ and k is the number of consecutive honest leaders needed.*

We prove in [18, Appendix A] that:

Theorem 5.2. *With a random leader election scheme, after GST with only omission faults, the expected number of rounds necessary to commit a block in BeeGees is $\frac{3n}{n-f}$*

Next, we simulate a scenario in which leaders are elected round-robin; we mark an operation as committed when there is sufficiently many honest leaders to satisfy the protocol's commit rule. In CLSO protocols, the number of rounds directly influences both latency and throughput. If a round has latency x, then commit latency for an operation will be $x * rounds$ while throughput is calculated by dividing the batch size by the expected commit latency. We write CHL(4) for Hotstuff (requires four consecutive leaders), CHL(3) for Fast-Hotstuff, Jolteon and DiemBFTv4, and finally AHL for our own protocol BeeGees. Figure 4 shows the resulting commit latency CDF. As expected, BeeGees achieves an expected commit latency of 4.5; CHL(3) requires ≈ 7 rounds. CHL(4) has worst expected performance, taking 12 rounds to commit. Worst-case observed commit latency is especially interesting: BeeGees has relatively low worst-case latency, with 18 rounds, while CHL(3) protocols have a worst-case commit time of 76. CHL(4) has seven times worst latency, with a worst-case commit time of 129 rounds.

6 CONCLUSION

This paper introduces BeeGees, the first CLSO protocol to guarantee that, after GST, the proposal of an honest leader will be committed after two honest views. In contrast, all other CLSO protocols require three (or four) consecutive honest leaders to commit a block. BeeGees observes that, to offer AHL, a protocol must guarantee sequentiality, and that sequentiality can only be enforced through careful use of Prepare messages. These are messages that are instead traditionally discarded during view changes by prior work. BeeGees's stronger liveness guarantee allows it to outperform other CLSO protocols by up to 4x. The full version of the paper [18] contains all appendices and proofs.

REFERENCES

[1] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2021. Brief Note: Fast Authenticated Byzantine Consensus. *CoRR* abs/2102.07932 (2021). https://doi.org/10.48550/ARXIV.2102.07932

[2] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2021. Good-Case Latency of Byzantine Broadcast: A Complete Categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event,

Italy) *(PODC'21)*. Association for Computing Machinery, New York, NY, USA, 331–341. https://doi.org/10.1145/3465084.3467899

[3] Ittai Abraham, Kartik Nayak, and Nibesh Shrestha. 2022. Optimal Good-Case Latency for Rotating Leader Synchronous BFT. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 217)*, Quentin Bramas, Vincent Gramoli, and Alessia Milani (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:19. https://doi.org/10.4230/LIPIcs.OPODIS.2021.27

[4] Mark Abspoel, Thomas Attema, and Matthieu Rambaud. 2020. Malicious Security Comes for Free in Consensus with Leaders. Cryptology ePrint Archive, Report 2020/1480. https://ia.cr/2020/1480.

[5] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. 2022. Twins: BFT Systems Made Robust. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 217)*, Quentin Bramas, Vincent Gramoli, and Alessia Milani (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:29. https://doi.org/10.4230/LIPIcs.OPODIS.2021.7

[6] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. 2020. Making Byzantine Consensus Live. In *34th International Symposium on Distributed Computing (DISC 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 179)*, Hagit Attiya (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:17. https://doi.org/10.4230/LIPIcs.DISC.2020.23

[7] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains.* Ph. D. Dissertation. University of Guelph. http://hdl.handle.net/10214/9769

[8] Vitalik Buterin and Virgil Griffith. 2017. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437* abs/1710.09437 (2017).

[9] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 173–186.

[10] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, Massachusetts) *(NSDI'09)*. USENIX Association, USA, 153–168.

[11] Shir Cohen, Rati Gelashvili, Lefteris Kokoris Kogias, Zekun Li, Dahlia Malkhi, Alberto Sonnino, and Alexander Spiegelman. 2022. Be Aware of Your Leaders. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers* (Grenada, Grenada). Springer-Verlag, Berlin, Heidelberg, 279–295. https://doi.org/10.1007/978-3-031-18283-9_13

[12] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2019. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234* (2019).

[13] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 34–50.

[14] Steve Drekic and Michael Z. Spivey. 2021. On the number of trials needed to obtain k consecutive successes. *Statistics & Probability Letters* 176, C (2021). https://doi.org/10.1016/j.spl.2021.109132

[15] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[16] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers* (Grenada, Grenada). Springer-Verlag, Berlin, Heidelberg, 296–315. https://doi.org/10.1007/978-3-031-18283-9_14

[17] Neil Giridharan, Heidi Howard, Ittai Abraham, Natacha Crooks, and Alin Tomescu. 2021. No-commit proofs: Defeating livelock in bft. *Cryptology ePrint Archive* (2021).

[18] Neil Giridharan, Florian Suri-Payer, Matthew Ding, Heidi Howard, Ittai Abraham, and Natacha Crooks. 2023. BeeGees: stayin' alive in chained BFT. arXiv:2205.11652 [cs.DC]

[19] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, USA, 568–580. https://doi.org/10.1109/DSN.2019.00063

[20] Lioba Heimbach and Roger Wattenhofer. 2022. SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance. *arXiv preprint arXiv:2203.11520* (2022).

[21] Mohammad M. Jalalzai, Jianyu Niu, and Chen Feng. 2020. Fast-HotStuff: A Fast and Resilient HotStuff Protocol. *CoRR* abs/2010.11454 (2020). arXiv:2010.11454 https://arxiv.org/abs/2010.11454

[22] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. 2021. Themis: Fast, strong order-fairness in byzantine consensus. *Cryptology ePrint Archive* (2021).

[23] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. 2020. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference*. Springer, 451–480.

[24] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems (TOCS)* 27, 4, Article 7 (Jan. 2010), 39 pages. https://doi.org/10.1145/1658357.1658358

[25] Dahlia Malkhi and Pawel Szalachowski. 2022. Maximal Extractable Value (MEV) Protection on a DAG. *arXiv preprint arXiv:2208.00940* (2022).

[26] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. 2021. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems* 1, 2 (Oct 2021). https://cryptoeconomicsystems.pubpub.org/pub/naor-cogsworth-synchronization.

[27] Jianyu Niu, Fangyu Gai, Mohammad M Jalalzai, and Chen Feng. 2021. On the performance of pipelined hotstuff. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–10.

[28] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2705–2718. https://doi.org/10.1145/3548606.3559361

[29] Xiao Sui, Sisi Duan, and Haibin Zhang. 2022. Marlin: Two-Phase BFT with Linearity. *Cryptology ePrint Archive* (2022).

[30] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 1–17.

[31] Aptos Team. [n.d.]. Aptos homepage. https://aptoslabs.com.

[32] Diem Team. 2021. *DiemBFT v4: State Machine Replication in the Diem Blockchain*. Technical Report. Diem. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf.

[33] Diem Team. [n.d.]. Diem homepage. https://www.diem.com.

[34] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

[35] Gengrui Zhang and Hans-Arno Jacobsen. 2021. Prosecutor: An efficient BFT consensus algorithm with behavior-aware penalization against Byzantine attacks. In *Proceedings of the 22nd International Middleware Conference*. 52–63.

[36] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine Ordered Consensus without Byzantine Oligarchy. (Nov. 2020), 633–649. https://www.usenix.org/conference/osdi20/presentation/zhang-yunhao