Bullshark: DAG BFT Protocols Made Practical

Alexander Spiegelman
sasha@aptoslabs.com
Aptos

Neil Giridharan
giridhn@berkeley.edu
UC Berkeley

Alberto Sonnino
alberto@mysternlabs.com
Mysten Labs

Lefteris Kokoris-Kogias
ekokoris@ist.ac.at
IST Austria

ABSTRACT

We present BullShark, the first directed acyclic graph (DAG) based asynchronous Byzantine Atomic Broadcast protocol that is optimized for the common synchronous case. Like previous DAG-based BFT protocols [19, 25], BullShark requires no extra communication to achieve consensus on top of building the DAG. That is, parties can totally order the vertices of the DAG by interpreting their local view of the DAG edges. Unlike other asynchronous DAG-based protocols, BullShark provides a practical low latency fast-path that exploits synchronous periods and deprecates the need for notoriously complex view-change mechanisms. BullShark achieves this while maintaining all the desired properties of its predecessor DAG-Rider [25]. Namely, it has optimal amortized communication complexity, it provides fairness and asynchronous liveness, and safety is guaranteed even under a quantum adversary.

In order to show the practicality and simplicity of our approach, we also introduce a standalone partially synchronous version of BullShark which we evaluate against the state of the art. The implemented protocol is embarrassingly simple (200 LOC on top of an existing DAG-based mempool implementation [19]). It is highly efficient, achieving for example, 125,000 transaction per second with a 2 seconds latency for a deployment of 50 parties. In the same setting the state of the art pays a steep 50% latency increase as it optimizes for asynchrony.

ACM Reference Format:
Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of ACM Conference, Los Angeles, CA, USA, November 2022 (Conference'22),* 17 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Ordering transactions in a distributed Byzantine environment via a consensus mechanism has become one of the most timely research areas in recent years due to the blooming Blockchain use-case. A recent line of work [8, 19, 21, 25, 33, 40] proposed an elegant way to separate between the distribution of transactions and the logic required to safely order them. The idea is simple. To propose transactions, parties send them in a way that forms a casual order among them. That is, messages contain blocks of transactions as well as references to previously received messages, which together form a *directed acyclic graph (DAG)*. Interestingly, the structure of the DAG encodes information that allow parties to totally order the DAG by locally interpreting their view of it without sending any extra messages. That is, once we build the DAG, implementing consensus on top of it requires *zero-overhead* of communication.

The pioneering work of Hashgraph [8] constructed an unstructured DAG, where each message refers to two previous ones, and used hashes of messages as local coin flips to totally order the DAG in asynchronous settings. Aleph [21] later introduced a structured round-based DAG and encoded a shared randomness in each round via a threshold signature scheme to achieve constant latency in expectation. The state of the art is DAG-Rider [25], which is built on previous ideas. Every round in its DAG has at most n vertices (one for each party), each of which contains a block of transactions as well as references (edges) to at least $2f + 1$ vertices in the previous round. Blocks are disseminated via reliable broadcast [11] to avoid equivocation and an honest party advances to the next round once it reliably delivers $2f + 1$ vertices in the current round. Note that building the DAG requires honest parties to broadcast vertices even if they have no transactions to propose. However, the edges of the DAG encodes the "voting" information that is sufficient to totally order all the DAG's vertices. So in this sense it is not different from other BFT protocols in which parties send explicit vote messages, which contain no transactions as well. Remarkably, by using the DAG to abstract away the communication layer, the entire edges interpretation logic of DAG-Rider to totally order the DAG spans over less than 30 lines of pseudocode.

DAG-Rider is an asynchronous Byzantine atomic broadcast (BAB), which achieves optimal amortized communication complexity ($O(n)$ per transaction), post quantum safety, and some notion of fairness (called Validity) that guarantees that every transaction proposed by an honest party is eventually delivered (ordered). To achieve optimal amortized communication DAG-Rider combines batching techniques with an efficient asynchronous verifiable information dispersal protocol [14] for the reliable broadcast building block. The protocol is post quantum safe because it does not rely on primitives that a quantum computer can brake for the safety properties. That is, a quantum adversary can prevent the protocol progress, but it cannot violate safety guarantees.

However, although DAG-based protocols have a solid theoretical foundation, they have multiple gaps before being realistically deployable in practise. First, they all optimize for the worst case

Conference'22, November 2022, Los Angeles, CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

asynchronous network assumptions and do not take advantage of synchronous periods, resulting to higher latency than existing consensus protocols [12, 41] in the good case. Second, they assume some impractical assumptions such as unbounded memory in order to preserve fairness. The only existing solution to this comes from Tusk [19] which uses a garbage collection mechanism but does not allow for quantifiable fairness even during periods of synchrony.

On the other hand, existing partially synchronous consensus protocols are designed as a monolith, where the leader of the protocol has to propose blocks of transactions in the critical path, resulting in performance bottlenecks and relatively low throughput as shown by Narwhal [19].

In this paper we optimize the mempool co-design mindset of DAGs to the partially synchronous case communication setting. First, we propose BullShark that preserves all the properties of DAG-Rider (including asynchronous worst case liveness), and in addition, introduces a fast path that exploits common-case synchronous network conditions. That is, BullShark is the first BAB protocol with optimal amortized communication complexity ($O(n)$ per transaction) and post quantum safety that is optimized for the common case. BullShark needs only 2 round-trips between commits during synchrony (thus a 75% improvement compared to DAG-Rider) and maintains a 6 round-trip expected latency in asynchronous executions (matching DAG-Rider).

Second, based on BullShark's fast path, we present an eventually synchronous variant of BullShark, which is the first partial synchronous consensus protocol that is completely embedded into a DAG. The protocol is fundamentally different from previous partially synchronous protocols since it is symmetric and *does not require a view-change* after a faulty leader. As a result, the resulting protocol is embarrassingly simple and extremely efficient, achieving 125k TPS and 2 second latency with 50 honest parties. As a final contribution, BullShark overcomes existing practical limitations of DAG-based protocols that do not allow for fairness together with garbage collection. BullShark garbage collects vertices belonging to old DAG rounds, but at the same time provide fairness during synchronous periods.

1.1 Technical challenges.

In order to design and implement BullShark we had to solve both theoretical and practical challenges. For example, the approach in current DAG-based protocols is to advance rounds as soon as enough messages in the current round are received ($2f+1$ for Aleph and DAG-Rider). Unfortunately, this approach cannot guarantee deterministic liveness during synchronous periods as required by the eventually synchronous variant of BullShark. This is because the adversary can, for example, reorder messages (within the synchrony bound) to make sure parties advance rounds before getting messages from the predefined leader. To solve this issue we needed to deconstruct the way partial synchronous protocols are designed and embed timeouts inside the DAG construction. The theoretical solution follows the DAG-Rider approach to advance rounds, but in addition, parties wait to receive some specific messages (e.g., from the predefined leader) or for a timeout to advance rounds.

A further challenge is to take advantage of a common-case synchronous network without sacrificing latency in the asynchronous worst case. To this end, BullShark introduces two types of votes - *steady-state* for the predefined leader and *fallback* for the random one. Similarly to DAG-Rider [25], BullShark rounds are grouped in *waves*, each of which consists of 4 rounds. Intuitively, each wave encodes a consensus logic. The first round of a wave has two potential leaders - a predefined steady-state leader and a leader that is chosen in retrospect by the randomness produced in the fourth round of the wave. To reduce latency in synchronous periods, the third round of a wave has a predefined leader as well and it takes two rounds to commit a steady-state leader. Based on their voting type, the vertices in the second round can potentially vote for the steady-state leader in the first round and vertices in the fourth round can potentially vote for the fallback leader in round one or the steady-state leader in round three. Importantly, the same vertex cannot vote for the fallback an a steady-state leaders in the same wave. A vertex's voting type is determined by whether or not its source (the party that broadcast it) committed a leader in the previous wave. This information is encoded in the DAG and since the DAG is built on top of a reliable broadcast abstraction, even byzantine parties cannot lie about their voting type. A nice property of BullShark is that it does not require a view change or view synchronization mechanisms to overcome faulty or slow leaders. Instead of a view change, BullShark uses the information encoded in the DAG to maintain safety. Since all parties agree on the causal history of every other party they have in the DAG, after a leader is committed each party locally "rides" the DAG (wave by wave) backwards to see which leader-vertices could have been committed by other parties. Synchronizing views is not required because (as we show in our proof) the DAG construction already provides it. If the first leader in a wave after GST is honest, then all parties are advancing to the third round of the wave roughly at the same time.

Finally, to evaluate BullShark we had to resolve some practical challenges. First, all previous theoretical solutions require unbounded memory to hold the entire DAG, and second, the reliable broadcast primitive we use to clearly describe BullShark (used in DAG-Rider and Aleph as well) is inefficient in the common-case. Fortunately, Narwhal [19] implemented a scalable DAG and dealt exactly with these problems. Their main motivation was to use DAG as an efficient blocks mempool for use from external protocols (e.g., Hotstuff). We, therefore, started from Narwhal's open source code base and adopt their efficient broadcast. Unfortunately, however, the Narwhal garbage collection mechanism directly conflicts with BullShark's mechanism to provide fairness. In fact, providing meaningful fairness for all honest parties seems to be impossible with bounded memory implementation in asynchronous network since every message can be delayed to after the relevant prefix of the DAG is garbage collected. To deal with this issue we relax our fairness requirement. That is, our bounded memory implementation of BullShark guarantees *timely fairness* only during synchronous periods. This means that after GST all messages by honest parties make it into the DAG in finite time and before the garbage collection. For all the other messages (before GST) we use Tusk's approach of retransmission, where guarantees can only be made for an unbounded execution.

In summary, this paper makes the following contributions:

- We propose BullShark, the first slow-path/fast-path DAG-based consensus protocol that achieves significantly lower latency than prior work. BullShark takes 2 rounds in the good case and 6 rounds (matching Tusk) in asynchrony
- We simplify BullShark to perform only in partial synchrony. This version of BullShark results in a significantly simpler partial synchronous consensus protocol than prior work (extra 200LOC vs 4000LOC of Hotstuff over a DAG [19]). BullShark additionally performs significantly better under faults making it the most performant and resilient partially synchronous protocol to date.
- We show how to build a practical DAG-based system that allows for garbage collection and provides timely fairness after GST, answering an open question of prior work [19, 25].

2 PRELIMINARIES

2.1 Model

We consider a peer to peer message passing model with a set of n parties $\Pi = \{p_1, \ldots, p_n\}$, and a *dynamic* adversary that can corrupt up to $f < n/3$ of them during an execution. We say that corrupted parties are *Byzantine* and all other parties are *honest*. Byzantine parties may act arbitrarily, while honest ones follow the protocol. We assume that the adversary is computationally bounded.

For the description of the protocol we assume that links between honest parties are reliable. That is, all messages among honest parties eventually arrive [1]. Moreover, for simplicity, we assume that recipients can verify the senders identities. We assume a known Δ and say that an execution of a protocol is *eventually synchronous* if there is a *global stabilization time (GST)* after which all messages sent among honest parties are delivered within Δ time. An execution is *synchronous* if GST occurs at time 0, and *asynchronous* if GST never occurs.

For the protocol analysis we are interested in the practical performance as well as theoretical complexity during synchronous and asynchronous periods, or alternatively, before and after the GST. To this end, we define consider the following scenarios:

- *Worst case condition*: asynchronous execution and f byzantine parties
- *Common case condition*: synchronous executions with no failures [2]

2.2 Building blocks

Similarly to DAG-Rider, we use the following known building blocks for our modular protocol presentation:

Reliable broadcast Each party p_k can broadcast messages by calling $r\_bcast_k(m, r)$, where m is a message and $r \in \mathbb{N}$ is a round number. Every party p_i has an output $r\_deliver_i(m, r, p_k)$, where m is a message, r is a round number, and p_k is the party that called the corresponding $r\_bcast_k(m, r)$. The reliable broadcast abstraction guarantees the following properties:

Agreement If an honest party p_i outputs $r\_deliver_i(m, r, p_k)$, then every other honest party p_j eventually outputs $r\_deliver_j(m, r, p_k)$.

Integrity For each round $r \in \mathbb{N}$ and party $p_k \in \Pi$, an honest party p_i outputs $r\_deliver_i(m, r, p_k)$ at most once regardless of m.

Validity If an honest party p_k calls $r\_bcast_k(m, r)$, then every honest party p_i eventually outputs $r\_deliver_i(m, r, p_k)$.

Global perfect coin An instance w, $w \in \mathbb{N}$, of the coin is invoked by party $p_i \in \Pi$ by calling $choose\_leader_i(w)$. This call returns a party $p_j \in \Pi$, which is the chosen leader for instance w. Let X_w be the random variable that represents the probability that the coin returns party p_j as the return value of the call $choose\_leader_i(w)$. The global perfect coin has the following guarantees:

Agreement If two honest parties p_i, p_j call $choose\_leader_i(w)$ and $choose\_leader_j(w)$ with respective return values p_1 and p_2, then $p_1 = p_2$.

Termination If at least $f+1$ honest parties call $choose\_leader(w)$, then every $choose\_leader(w)$ call eventually returns.

Unpredictability As long as less than $f+1$ honest parties call $choose\_leader(w)$, the return value is indistinguishable from a random value except with negligible probability ϵ. Namely, the probability pr that the adversary can guess the returned party p_j of the call $choose\_leader(w)$ is $pr \leq \Pr[X_w = p_j] + \epsilon$.

Fairness The coin is fair, i.e., $\forall w \in \mathbb{N}, \forall p_j \in \Pi$: $\Pr[X_w = p_j] = 1/n$.

Implementation examples that use PKI and a threshold signature scheme [10, 29, 34] can be found in [13, 30]. See DAG-Rider for more details on how a coin implementation can be integrated into the DAG construction. It is important to note that the above mentioned implementations satisfy Agreement, Termination, and Fairness with information theoretical guarantees. That is, the assumption of a computationally bounded adversary is required only for the unpredictability property. As we later prove, the unpredictability property is only required for Liveness. Therefore, since similarly to DAG-Rider generating randomness is the only place where cryptography is used, the Safety properties of BullShark are post-quantum secure.

2.3 Problem Definition

Following DAG-Rider [25], our result focuses on the *Byzantine Atomic Broadcast (BAB)* problem. To avoid confusion with the events of the underlying reliable broadcast abstraction, the broadcast and deliver events of BAB are $a\_bcast(m, r)$ and $a\_deliver(m, r, p_k)$, respectively, where m is a message, $r \in \mathbb{N}$ is a sequence number, and $p_k \in \Pi$ is a party. The purpose of the sequence numbers is to distinguish between messages broadcast by the same party. We assume that each party broadcasts infinitely many messages with consecutive sequence numbers.

Definition 2.1 (Byzantine Atomic Broadcast). Each honest party $p_i \in \Pi$ can call $a\_bcast_i(m, r)$ and output $a\_deliver_i(m, r, p_k)$, $p_k \in \Pi$. A Byzantine Atomic Broadcast protocol satisfies reliable broadcast (agreement, integrity, and validity) as well as:

Total order If an honest party p_i outputs $a\_deliver_i(m, r, p_k)$ before $a\_deliver_i(m', r', p'_k)$, then no honest party p_j outputs $a\_deliver_j(m', r', p'_k)$ before $a\_deliver_j(m, r, p_k)$.

[1] We address this issues from a practical point of view in our implementation.

[2] Same analysis apply to eventually synchronous failure-free executions after GST.

Note that the above definition is agnostic to the network assumptions. However, in asynchronous executions, due to the FLP result [20], BAB cannot be solved deterministically and therefore we relax the validity property to hold with probability 1 in this case. Moreover, the validity property cannot be satisfied in asynchronous executions with bounded memory implementation. Therefore, as we discuss more in Section 6, for the practical version of this problem, we require validity to be satisfied only after GST in eventually synchronous executions.

Note that the BAB abstraction captures the core consensus logic in permissioned Blockchain systems as it provides a mechanism to propose blocks of transactions and totally order them. Moreover, similarly to Hyperledger [7], it supports a separation between the total order mechanism and transaction execution. Transaction validation can therefore be done as part of the execution [7] before applying to the SMR.

3 DAG CONSTRUCTION

In this section we describe our DAG construction and explain how it is different from the one in DAG-Rider [25]. In a nutshell, DAG-Rider is a fully asynchronous atomic broadcast protocol and thus rounds in its DAG advance in network speed as soon as $2f+1$ nodes from the current round are delivered. Here, we are interested in a protocol that achieves better latency in synchronous periods and thus introduce timeouts into the DAG construction. It is important to note that despite the timeouts, our DAG still advances in network speed when the leader is honest. We present the background, structures, and basic utilities we borrow from DAG-Rider in Section 3.1. We describe our DAG construction in Section 3.2.

3.1 Background

We use a DAG to abstract the communication layer among parties and enable the establishment of common knowledge. Each vertex in the DAG represents a message disseminated via reliable broadcast from a single party, containing, among other data, references to previously broadcast vertices. Those references are the edges of the DAG. Each honest process maintains a local copy of the DAG and different honest parties might observe different versions of it (depending on the order in which they deliver the vertices). Nevertheless, the reliable broadcast prevents equivocation and guarantees that all honest parties eventually deliver the same messages, hence their views of the DAG eventually converge.

The DAG data types and and basic utilities are specified in Algorithm 1. For each party p_i, we denote p_i's local view of the DAG as DAG_i, which is represented by an array of sets of vertices $DAG_i[]$. Vertexes are created via the $create\_new\_vertex(r)$ procedure. Each vertex in the DAG is associated with a unique round number r and a the party who generated and reliably broadcast it (the source). In addition, each vertex v contains a block of transactions that were previously *a_bcast* by the BAB protocol that is implemented on top of the DAG and two sets of outgoing edges. The set *strong edges* contains at least $2f+1$ references to vertexes associated with round $r-1$ and the set *weak edges* contains up to f references to vertices in rounds $< r-1$ such that otherwise there is no path from v to them. As explained in the next sections, strong edges are used for Safety and weak edges make sure we eventually include all vertices in the total order, to satisfy BAB's validity property.

The entry $DAG_i[r]$ for $r \in \mathbb{N}$ stores a set of vertices associated with round r that p_i previously delivered. By the reliable broadcast, each party can broadcast at most 1 vertex in each round and thus $|DAG_i[r]| \leq n$.

The procedures $path(v,u)$ and $strong\_path(v,u)$ get two vertexes and check if there is a path from v to u. The difference between them is that $path(v,u)$ considers all edges while $strong\_path(v,u)$ only considers the strong ones.

The procedure $get\_fallback\_vertex\_leader$ gets a wave number, computes the randomly elected leader of the wave and then returns the vertex that the elected leader broadcast in the first round of the wave, if it is included in the DAG. Otherwise, returns $\perp$. Similarly, the procedures $get\_first\_steady\_vertex\_leader$ and $get\_second\_steady\_vertex\_leader$ return the vertices broadcast by the first and second predefined leaders of the wave, respectively. We assume a predefined and known to all parties mapping waves to steady-state leaders.

3.2 Our DAG protocol

A detailed pseudocode is given in Algorithm 2. Each party p_i maintains three local variables: *round* stores the last round in which p_i broadcast a vertex, *buffer* stores vertices that where reliably delivered but not yet added to the DAG, and *wait* is an Boolean that indicate whether the timeout for the current round has already expired. Each party p_i is constantly trying to advance rounds and calling the high-level BAB protocol to totally order all the vertices in its DAG. When p_i advances its round, it broadcast its vertex for this round and start a timeout.

Our DAG protocol is triggered by one of two events: a vertex delivery (via reliable broadcast) or a timeout expiration. Once a party p_i delivers a vertex it first checks if the vertex is legal, i.e., (1) the source and round must match the reliable broadcast instance to prevent equivocation, and (2) the vertex must has at least $2f+1$ strong edges. Then, p_i checks if the vertex is ready to be added to the DAG by calling $try\_add\_to\_DAG$. The idea is to make sure that the causal history of a vertex is always available in the DAG. Therefore, a vertex is added to the DAG only if all the vertices it includes as references are already delivered. If this is not yet the case, the vertex is added to a *buffer* for a later retry. Once a vertex v is added to the DAG, the high-level BAB protocol is invoked, via the $try\_ordering(v)$ interface, to check if more vertices can now be totally ordered.

We next describe the conditions for advancing rounds. Note that since DAG-Rider only cares about the asynchronous case, rounds are advanced as soon as $2f+1$ vertices in the current round are delivered. We, in contrast, optimize for the common case conditions and thus have to make sure that parties do not advance rounds too fast. Otherwise, the adversary can prevent honest parties from committing steady-state leaders since it controls which $2f+1$ vertexes parties deliver first even after GST. Therefore, we keep the DAG-Rider necessary condition (in $try\_advance\_round$) but extend it to make sure that honest steady-state leaders are committed in network speed after GST.

Algorithm 1 Data structures and basic utilities for party p_i

```
Local variables:
    struct vertex v:                                                    ▷ The struct of a vertex in the DAG
        $v.round$ - the round of $v$ in the DAG
        $v.source$ - the party that broadcast $v$
        $v.block$ - a block of transactions
        $v.strongEdges$ - a set of vertices in $v.round - 1$ that represent strong edges
        $v.weakEdges$ - a set of vertices in rounds $< v.round - 1$ that represent weak edges
    $DAG_i[\,]$ - An array of sets of vertices, initially:
        $DAG_i[0] \leftarrow$ predefined hardcoded set of $2f + 1$ “genesis” vertices
        $\forall j \geq 1$: $DAG_i[j] \leftarrow \{\}$
    blocksToPropose - A queue, initially empty, $p_i$ enqueues valid blocks of transactions from clients

procedure path(v, u)                    ▷ Check if exists a path consisting of strong and weak edges in the DAG
    return exists a sequence of $k \in \mathbb{N}$, vertices $v_1, v_2, \ldots, v_k$ s.t.
        $v_1 = v$, $v_k = u$, and $\forall i \in [2..k]$: $v_i \in \bigcup_{r\geq 1} DAG_i[r] \wedge (v_i \in v_{i-1}.weakEdges \cup v_{i-1}.strongEdges)$

procedure strong_path(v, u)             ▷ Check if exists a path consisting of only strong edges in the DAG
    return exists a sequence of $k \in \mathbb{N}$, vertices $v_1, v_2, \ldots, v_k$ s.t.
        $v_1 = v$, $v_k = u$, and $\forall i \in [2..k]$: $v_i \in \bigcup_{r\geq 1} DAG_i[r] \wedge v_i \in v_{i-1}.strongEdges$

procedure create_new_vertex(round)
    wait until ¬blocksToPropose.empty()
    $v.round \leftarrow round$
    $v.source \leftarrow p_i$
    $v.block \leftarrow$ blocksToPropose.dequeue()
    $v.strongEdges \leftarrow DAG[round - 1]$
    set_weak_edges(v, round)
    return v

procedure set_weak_edges(v, round)      ▷ Add edges to orphan vertices
    $v.weakEdges \leftarrow \{\}$
    for $r = round - 2$ down to 1 do
        for every $u \in DAG_i[r]$ s.t. $\neg path(v, u)$ do
            $v.weakEdges \leftarrow v.weakEdges \cup \{u\}$

procedure get_fallback_vertex_leader(w)
    $p \leftarrow choose\_leader_i(w)$
    return $get\_vertex(p, 4w - 3)$

procedure get_first_steady_vertex_leader(w)
    $p \leftarrow get\_first\_predefined\_leader(w)$
    return $get\_vertex(p, 4w - 3)$

procedure get_second_steady_vertex_leader(w)
    $p \leftarrow get\_second\_predefined\_leader(w)$
    return $get\_vertex(p, 4w - 1)$

procedure get_vertex(p,r)
    if $\exists v \in DAG[r]$ s.t. $v.source = p$ then
        return v
    return ⊥
```

We distinguish between slow and up-to-date parties. As mentioned in the introduction, BullShark does not require an external view-synchronization mechanism for slow parties. Instead, once p_i delivers $2f + 1$ vertices in a round $r > round$, p_i jumps forward to round r, broadcasts a vertex in round r, and starts a new timeout.

For the up-to-date parties we need to be more careful. As we explained more in the next section, each wave has a steady-state leader in the first round and a steady-state leader in the third one. Intuitively, the vertices of these leaders are interpreted as "proposals" and the vertices in immediately following rounds with strong edges to the leaders’ vertices are interpreted as "votes". In addition, each party can vote for the steady-state leaders in a wave only if its voting type is steady-state for this wave. To make sure all honest parties get a chance to vote for steady state leaders, an up-to-date honest party p_i will try to advance (via *try_advance_round*) to the second and forth rounds of a wave only if (1) the timeout for this round expired or (2) p_i delivered a vertex from the wave predefined first and second steady-state leader, respectively. Similarly, we need to make sure the adversary cannot prevent honest parties from collecting enough votes to commit an honest leader after GST. Therefore, before trying to advance (via *try_advance_round*) to the third round a wave or the first round of the next wave, p_i waits for either the timeout expiration or to deliver $2f + 1$ vertices in the current round with steady-state voting type and strong edges to the first and second steady-leader, respectively. In Section 7.2, we prove that after GST timeouts never expire for honest leaders and the DAG advances in network speed.

4 THE BULLSHARK PROTOCOL

In this section we present a detailed description of BullShark. Similarly to DAG-Rider [25], the ordering logic of BullShark requires no communication on top of building the DAG. Instead, each party observes its local copy of the DAG and totally order its vertices by interpreting the edges as "votes". In order to optimize for the common case conditions while guaranteeing liveness under worst case asynchronous conditions, BullShark has two types of leaders: *steady-state* and *fallback*. The main challenge in designing BullShark is the interplay between them as we need to make sure parties cannot vote for both types at the same round. We divide the protocol description into two parts. In Section 4.1 we describe the commit rule of each leader, and in Section 4.2 we explain how parties totally order leaders’ causal histories. In Section 5 we preset an eventually synchronous version of BullShark and in Section 6 we discuss the details of our garbage collection mechanism. We give full formal proofs for both versions in Section 7.

4.1 Voting Types

Similarly to DAG-Rider, to interpret the DAG, each party p_i divides its local view of the DAG, DAG_i, into waves of 4 rounds each. Unlike DAG-Rider, which has one potential leader in every wave, BullShark has three. One *fallback* leader in the first round of each wave, which is elected retrospectively via the randomness produced in the forth round of the wave (as in DAG-Rider), and two predefined *steady-state* leaders in the first and third rounds of each wave.

Algorithm 2 DAG construction, protocol for process p_i

```
Local variables:
    $round \leftarrow 1$; $buffer \leftarrow \{\}$; $wait \leftarrow true$

upon $r\_deliver_i(v, r, p)$ do
    if $v.source = p \wedge v.round = r \wedge |v.strongEdges| \geq 2f + 1$ then
        if $\neg try\_add\_to\_dag(v)$ then
            $buffer \leftarrow buffer \cup \{v\}$
        else
            for $v \in buffer : v.round \leq r$ do
                $try\_add\_to\_dag(v)$
        if $r = round$ then
            $w \leftarrow \lceil r/4 \rceil$                              ▷ steady state wave number
            if $r \bmod 4 = 1 \wedge (\neg wait \vee \exists v \in DAG[r] : v.source = get\_first\_steady\_vertex\_leader(w))$ then
                $try\_advance\_round()$
            if $r \bmod 4 = 3 \wedge (\neg wait \vee \exists v \in DAG[r] : v.source = get\_second\_steady\_vertex\_leader(w))$ then
                $try\_advance\_round()$
            if $r \bmod 4 = 0 \wedge (\neg wait \vee \exists U \subseteq DAG[r] : |U| = 2f + 1$ and $\forall u \in U, u.source \in steadyVoters[w]) \wedge strong\_path(u, get\_second\_steady\_leader(w))$ then
                $try\_advance\_round()$
            if $r \bmod 4 = 2 \wedge (\neg wait \vee \exists U \subseteq DAG[r] : |U| = 2f + 1$ and $\forall u \in U, u.source \in steadyVoters[w]) \wedge strong\_path(u, get\_first\_steady\_leader(w))$ then
                $try\_advance\_round()$

upon timeout do
    $wait \leftarrow false$
    $try\_advance\_round()$

procedure try_add_to_dag($v$)
    if $\forall v' \in v.strongEdges \cup v.weakEdges : v' \in \bigcup_{k \geq 1} DAG[k]$ then
        $DAG[v.round] \leftarrow DAG[v.round] \cup \{v\}$
        if $|DAG[v.round]| \geq 2f + 1 \wedge v.round > round$ then
            $round \leftarrow v.round$; start timer; $wait \leftarrow true$   ▷ Synchronize waves
            $broadcast\_vertex(v.round)$
        $buffer \leftarrow buffer \setminus \{v\}$
        $try\_ordering(v)$
        return true
    return false

procedure try_advance_round()
    if $|DAG[round]| \geq 2f + 1$ then
        $round \leftarrow round + 1$; start timer; $wait \leftarrow true$
        $broadcast\_vertex(round)$

procedure broadcast_vertex(r)
    $v \leftarrow create\_new\_vertex(r)$
    $try\_add\_to\_dag(v)$
    $r\_bcast_i(v, r)$
```

In the common case, during synchronous periods, both steady-state leaders are committed in each wave, meaning that it takes two rounds on the DAG to commit a leader. During asynchronous periods, each fallback leader is committed with probability of at least 2/3. Meaning that during asynchrony, a fallback leader is committed every 6 rounds in expectation and BullShark has liveness with probability 1.

A nice property of the common case execution of BullShark is that it does not require external view-change and view-synchronization mechanisms. When switching from asynchrony to synchrony, the first two rounds of each wave make sure that if the first leader is honest then all honest parties start the third round roughly at the same time. View-change is not required because the DAG encodes all the information needed for safety. In particular, parties can see what information other parties had when they interpreted the DAG, and decide accordingly.

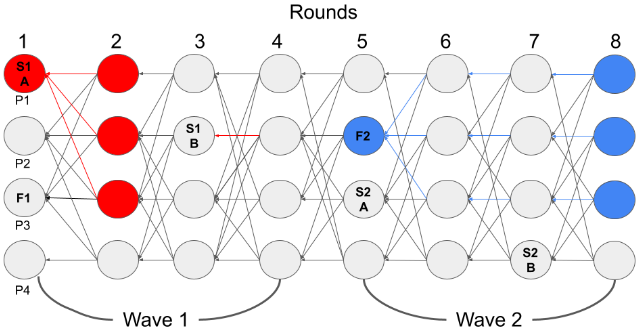

Figure 1: Illustration of the DAG at party P1. The columns represent the round numbers and the rows are all the vertices from a particular party (P1, P2, P3, P4 top to bottom). S1A denotes the first steady state leader of wave 1 (in round 1), and S1B denotes the second steady state leader of wave 1 (in round 3). F1 denotes the fallback leader of wave 1 (in round 1). All parties start off with a steady state vote type in wave 1. In round 2, P1 observes 3 (2f + 1) steady state votes for S1A (denoted in red), so P1 commits S1A. In round 4, P1 only observes 1 vote for the second steady state leader S1B, so P1 does not commit S1B. Since P1 does not commit the second steady state leader, it has a fallback vote type in wave 2. From the DAG in round 5, P1 also observes that P2, P3, P4 did not commit S1B, so all parties have a fallback vote type in wave 2. Thus S2A and S2B (the first and second steady state leaders in wave 2 respectively) cannot be committed since all vote types are fallback. In round 8, P1 observes 3 (2f + 1) fallback votes for the fallback leader F2 (denoted in blue), so P1 commits F2. Once P1 commits F2, it checks to see whether any previous leader it did not commit, could have been committed. In round 4, P1 only observes 1 steady state vote for S1B (less than f + 1), so it does not commit S1B since if it would have been committed by some party then P1 would observed at least $f + 1$ votes.

The pseudocode appears in Algorithm 3. The procedure try_ordering is called every time a new vertex is added to the DAG. Since BullShark has two types of leaders in each wave, we need to ensure that fallback and steady-state leaders are never committed in the same wave. To this end, parties cannot vote for both types of leaders in the same wave. That is, every party is assigned with a voting type in every wave that is either fallback or steady-state. When a party p_i interprets its local copy of the DAG it keeps track of other parties voting types in *steadyVoters[w]* and *fallbackVoters[w]*, where w is a wave number.

Intuitively, a party is in *steadyVoters[w]* if it has committed either the second steady-state or the fallback leader in wave $w - 1$. Specifically, party p_i determines p_j's voting type in wave w when it delivers p_j's vertex v in the first round of wave w, which triggers the call to the determine_party_vote_type procedure. If the causal history of v has enough information to commit one of these leaders, then p_i determines p_j's voting type as steady-state, otherwise, as fallback. By the properties of reliable broadcast, all parties see the same causal history of vertex v, and thus agree on p_j's voting type in round w (even Byzantine parties cannot lie about their voting type).

To commit a leader in wave $w - 1$ based on a vertex v in the first round of a wave w, p_i considers the set of vertices pointed by v's

Algorithm 3 BullShark part 1: p_i's alg. to update parties vote type

```
Local variables:
    steadyVoters[1] ← Π; fallbackVoters[1] ← {}
    For every $j > 1$, steadyVoters[j], fallbackVoters[j] ← {}

upon $a\_bcast_i(b, r)$ do
    blocksToPropose.enqueue(b)

procedure try_ordering(v)
    $w \leftarrow \lceil v.round/4 \rceil$
    votes ← v.strongEdges
    if v.round mod 4 = 1 then                          ▷ first round of a wave
        determine_party_vote_type(v.source, votes, w)
    else if v.round mod 4 = 3 then
        try_steady_commit(votes, get_first_steady_vertex_leader(w), w)

procedure determine_party_vote_type(p, votes, w)
    $v_s$ ← get_second_steady_vertex_leader(w-1)
    $v_f$ ← get_fallback_vertex_leader(w-1)
    if try_steady_commit(votes, $v_s$, $w-1$) ∨ try_fallback_commit(votes, $v_f$, $w-1$) then
        steadyVoters[w] ← steadyVoters[w] ∪ {p}
    else
        fallbackVoters[w] ← fallbackVoters[w] ∪ {p}

procedure try_steady_commit(votes, v, w)
    if $|\{v' \in votes : v'.source \in staedyVoters[w] \wedge strong\_path(v', v)\}| \geq 2f + 1$ then
        commit_leader(v)
        return true
    return false

procedure try_fallback_commit(votes, v, w)
    if $|\{v' \in votes : v'.source \in fallbackVoters[w] \wedge strong\_path(v', v)\}| \geq 2f + 1$ then
        commit_leader(v)
        return true
    return false
```

strong edges as potential "votes". Note that these vertices belong to wave $w - 1$ and each of them has a voting type that was already previously determined by p_i. To commit the fallback leader of wave $w - 1$, at least of $2f + 1$ out of the potential votes must have strong paths to the leader and a fallback voting type. Similarly, to commit the second steady-state leader of wave $w - 1$, at least $2f + 1$ out of the potential votes must to have strong paths to the leader and steady-state voting type. Committing the first steady-state leader of a wave is similar but in this case the strong edges of a vertex in the third round of the wave are considered as potential votes. Note that since even a Byzantine party cannot lie about its voting type, quorum intersection guarantees that leaders with different types cannot be committed in the same wave. This is the reason we ask for $2f + 1$ strong paths unlike Tusk where $f + 1$ strong paths are sufficient for safety. As we describe next, when a leader v is committed then the procedure commit_leader is called to totally order v's causal history.

4.2 Ordering The DAG

So far we described the wave commit rules and how parties use them to determine other parties voting types. Next we describe how we totally order the DAG. The pseudocode appears in Algorithm 4. Once a party p_i commits a (steady-state or fallback) leader vertex v it calls $commit\_leader(v)$. To totally order the causal history of v, p_i first tries to commit previous leaders for which the commit rule in its local copy of the DAG was not satisfied. To do this, p_i traverses back the rounds of its DAG until the last round in which it committed a leader and check whether it is possible that other honest parties committed leaders in these rounds based on their local copy of the DAG. If p_i encounters such a leader, it orders it before v. Note that this part is much trickier than in DAG-Rider since BullShark has three potential leaders in every wave.

Algorithm 4 BullShark part 2: the commit alg. for party p_i

```
Local variables:
    committedRound ← 0
    deliveredVertices ← {}
    leaderStack ← initialize empty stack

procedure commit_leader(v)
    leaderStack.push(v)
    $r \leftarrow v.round - 2$    ▷ There is a potential leader to commit every two rounds
    while $r > committedRound$ do
        $w \leftarrow \lceil r/4 \rceil$
        ssPotentialVotes ← $\{v' \in DAG_i[r+1] \mid strong\_path(v, v')\}$
        if r mod 4 == 1 then                     ▷ two potential leaders in this round
            $v_s \leftarrow get\_first\_steady\_vertex\_leader(w)$
            $v_f \leftarrow get\_fallback\_vertex\_leader(w)$
            ssVotes ← $\{v' \in ssPotentialVotes : v'.source \in$
                        $steadyVoters[w] \wedge strong\_path(v', v_s)\}$
            if $v.round = r + 2$ then
                fbVotes ← {}      ▷ fallback leader could not be committed since
                                  there at least $2f + 1$ steady-state vote types in this wave
            else
                fbPotentialVotes ← $\{v' \in DAG_i[r+3] \mid strong\_path(v, v')\}$
                fbVotes ← $\{v' \in fbPotentialVotes : v'.source \in$
                            $fallbackVoters[w] \wedge strong\_path(v', v_f)\}$
        else                                                      ▷ r mod 4 == 3
            $v_s \leftarrow get\_second\_steady\_vertex\_leader(w)$
            ssVotes ← $\{v' \in ssPotentialVotes : v'.source \in$
                        $steadyVoters[w] \wedge strong\_path(v', v_s)\}$
            $v_f \leftarrow \bot$; fbVotes ← {}
        if $|ssVotes| \geq f + 1 \wedge |fbVotes| < f + 1$ then
            $leadersStack.push(v_s)$
            $v \leftarrow v_s$
        if $|ssVotes| < f + 1 \wedge |fbVotes| \geq f + 1$ then
            $leadersStack.push(v_f)$
            $v \leftarrow v_f$
        $r \leftarrow r - 2$
    committedRound ← $v.round$
    $order\_vertices()$

procedure $order\_vertices()$
    while ¬leadersStack.isEmpty() do
        v ← leadersStack.pop()
        verticesToDeliver ← $\{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v, v') \wedge v' \notin$
                            $deliveredVertices\}$
        for every $v' \in verticesToDeliver$ in some deterministic order do
            output $a\_deliver_i(v'.block, v'.round, v'.source)$
            deliveredVertices ← deliveredVertices ∪ {v'}
```

By quorum intersection and the non-equivocation property of the DAG, if some party commits either a fallback or a steady-state leader by seeing $2f + 1$ votes, then all other parties see at least $f + 1$ of these votes. Moreover, since a party cannot vote for both types of leaders in the same wave, if p_i sees $f + 1$ votes for the fallback (steady-state) leader, then no party could have committed the steady-state (fallback) leader since in this case there are at most $2f$ votes with steady-state (fallback) type.

To make sure p_i orders the leaders that precedes v consistently with the other parties, we need to make sure that parties consider the same potential votes when deciding whether to order one of them. To this end, to decide whether to order a steady-state leader v',

p_i sets the potential votes to be all the vertices in round $v'.round+1$ in its DAG such that there is a strong path between the last leader p_i previously ordered and v'. For a fallback leader v', the potential votes are set in a similar way but round $v'+3$ is used instead of $v'+1$ to be consistent with the commit rule.

After computing the potential votes, p_i checks if one of the leaders in the round it is currently traversing could be committed by other honest parties. First, p_i checks the potential votes type and the existence of strong paths to the leaders to determines the sets of votes for the steady-state and fallback leaders. Note that the set of votes for the fallback leader is empty in rounds without a fallback leader or if a steady-state leader was already committed in this wave. Then, p_i checks if one of the leaders u in the round has at least $f+1$ votes while the other has at most f. If this is the case p_i orders u by pushing it to the leader's stack *leaderStack* and continues its traversal to the next rounds to check if there are leaders to order before u. Otherwise, p_i skips the leaders of the current round as it is guaranteed that none if them could have been committed.

As we prove in Section 7, all honest parties order the same leaders and in the same order. All that is left is to apply some deterministic rule to order their causal histories one by one. Therefore, after committing a leader v (and finishing ordering all leaders that proceeds v for which the commit rule was not satisfied), party p_i calls $order\_vertex()$. This function goes over the ordered leaders one by one, and for each of them delivers, by some deterministic order, all the blocks in the vertices in it causal history (strong and weak edges) that have not yet been delivered.

5 EVENTUALLY SYNCHRONOUS BULLSHARK

In this section we present an eventually synchronous version of the Bullshark protocol. This protocol is embarrassingly simple, and as we demonstrate in Section 9, very efficient. To the best of our knowledge, this is the first eventually synchronous BFT protocol that does not require view-change or view-synchronization mechanism. In a nutshell, there are no fallback leaders in the eventually synchronous version of BullShark. Instead, parties keep trying to commit the steady-state leaders. The pseudocode, which overwrites the *try_ordering* procedure, appears in Algorithm 5 (Note that some procedures from previous Algorithms are called). In section 7.2 we give a formal proof of Safety and Liveness after GST. In a nutshell, the safety proof has a similar proof structure as BullShark with fallback, and for liveness we show that after GST two consecutive honest predefined leaders guarantee that the second leader will be committed by all honest parties. In particular, we show that if the first leader of wave w is honest, then all honest parties advance to the third round of w roughly at the same time. Moreover, if the second leader is honest than all honest parties will wait for him before advancing to the forth round, and thus all honest will see at least $2f+1$ votes for the second leader in w and commit it.

6 GARBAGE COLLECTION IN BULLSHARK

One of the main practical challenges and a potential reason that DAG-based BFT protocols are not yet widely deployed is the need for unbounded memory to guarantee validity and fairness. In other words, the question of how to satisfy fairness and at the same time garbage collect old parts of the DAG from the working memory of the system.

Algorithm 5 Eventually synchronous BullShark: alg. for party p_i.

```
Local variables:
    committedRound ← 0
    leaderStack ← initialize empty stack

procedure try_ordering($v$)
    $w \leftarrow \lceil v.round/4 \rceil$
    votes ← $v$.strongEdges
    if $v$.round mod 4 = 1 then        ▹ try committing second leader of prev wave
        try_commit(votes, get_second_steady_vertex_leader(w-1))
    else if $v$.round mod 4 = 3 then   ▹ try committing first leader of this wave
        try_commit(votes, get_first_steady_vertex_leader(w))

procedure try_commit($votes$, $v$)
    if $|\{v' \in votes : strong\_path(v', v)\}| \geq 2f+1$ then
        $commit\_leader(v)$

procedure commit_leader($v$)
    $leadersStack.push(v)$
    $r \leftarrow v.round - 2$
    while r > committedRound do
        $w \leftarrow \lceil v.round/4 \rceil$
        if $r\ mod\ 4 == 1$ then
            $v_s \leftarrow get\_first\_steady\_vertex\_leader(w)$
        else                                              ▹ $r\ mod\ 4 == 3$
            $v_s \leftarrow get\_second\_steady\_vertex\_leader(w)$
        if $strong\_path(v, v_s)$ then
            $leadersStack.push(v_s)$
            $v \leftarrow v_s$
        $r \leftarrow r - 2$
    committedRound ← $v.round$
    $order\_vertices()$                                   ▹ see Algorithm 4
```

For example, HashGraph [8] constructs an unstructured DAG, and thus has to keep in memory the entire prefix of the DAG in order to verify the validity of new blocks. DAG-Rider[25], Aleph [21], and Narwhal [19] use a round-based structured DAG, but do not provide a solution to the aforementioned question. The only DAG-based BFT we are aware of that proposed a garbage collection mechanism is Narwhal [19]. Their mechanism uses the consensus decision in order to agree what rounds in the DAG can be cleaned. However their protocol sacrifices the Validity (fairness) property of the BAB problem. It does not provide fairness to all parties since blocks of slow parties can be garbage collected before they have a chance to be totally ordered. DAG-Rider[25], on the other hand, make use of weak links to refer to yet unordered blocks in previous rounds, which guarantees that every block is eventually ordered. The solution works well in theory, but it is unclear how to garbage collect it.

In fact, through our investigation we realized that providing the BAB's validity (fairness) property with bounded memory in fully asynchronous executions is impossible since blocks of honest parties can be arbitrarily delayed. Similarly to the core observation in the FLP [20] impossibility result, in asynchronous settings, it is impossible to distinguish between faulty parties that will never broadcast a block and slow parties for which we need to wait before garbage collecting old rounds.

Fairness after GST. In the BullShark implementation we propose a practical alternative. We maintain bounded memory at the cost of providing fairness only after GST. What we need is a $\diamond P$ failure detector [18, 28] which will be strong and complete after GST letting us garbage collect rounds even if we did not get vertices from all parities (i.e., we do not need to wait forever for faulty parties). We do it by leveraging the structure of our DAG and introducing the notion of timestamp as described below. Formally, our implementation of BullShark maintains bounded memory and satisfies the following:

Definition 6.1. If an honest party p_k calls $r\_bcast_k(m, r)$ after GST, then every honest party p_i eventually outputs $r\_deliver_i(m, r, k)$.

For the garbage collection mechanism we add a timestamp for every vertex. That is, an honest party specify in $v.ts$ the time when it broadcast its vertex v. In addition, parties maintain a garbage collection round, *GCround*, and never add vertices to the DAG in rounds below it. Note that the latency of the reliably broadcast building block we use is bounded after GST, but depends on the specific implementation. For the protocol description we assume that the time it takes to reliably broadcast a message after GST is Δ. The pseudocode, in which we describe how to change the function order_vertices that is used by both versions of BullShark, appears in Algorithm 6. The idea is simple. For every leader v we order, we assign a timestamp ts, which is computed as the median of all the timestamp of v' parents (i.e., v's strong edges). Then, while traversing v's causal history to find vertices to order, we compute a timestamp for every round in a similar way (the median of timestamps of the vertices in this round). If the difference between the timestamp is above 3Δ the round is garbage collected.

Since by the properties of the underling reliable broadcast all parties agree on the causal histories of the leaders, once parties agree which leaders to order they also agree what rounds to garbage collect. Therefore, the garbage collection mechanism preserves the safety and liveness properties we prove in Section 7. Below we argue that when announced with the above garbage collection, BullShark satisfies Definition 6.1 while preserving bounded memory.

Algorithm 6 Garbage collection. Algorithm for party p_i.

```
Local variables:
    GCround ← 0
procedure $order\_vertices()$
    while $\neg leadersStack.isEmpty()$ do
        $v \leftarrow leadersStack.pop()$
        if $v.round > 1$ then
            $parents \leftarrow \{u \in DAG_i[v.round - 1] \mid path(v, u)\}$
            $leaderTS \leftarrow median(\{v.ts \mid v \in parents\})$
            $verticesToDeliver \leftarrow parents \cup \{v\}$
        else
            $verticesToDeliver \leftarrow \{v\}$
        $r \leftarrow GCround + 1$
        while $r < v.round - 1$ do
            $candidates \leftarrow \{u \in DAG_i[r] \mid path(v, u)\}$
            $candidatesTS \leftarrow median(\{v.ts \mid v \in candidates\})$
            $verticesToDeliver \leftarrow verticesToDeliver \cup candidates \setminus deliveredVertices$
            if $leaderTS$ - $candidatesTS > 3\Delta$ then
                $GCround \leftarrow r$
                $DAG_i[r] \leftarrow \{\}$                ▷ garbage collect old rounds
            $r \leftarrow r + 1$
        for every $v' \in verticesToDeliver$ do        ▷ in some deterministic order
            output $a\_deliver_i(v'.block, v'.round, v'.source)$
            $deliveredVertices \leftarrow deliveredVertices \cup \{v'\}$
```

Bounded memory. In Section 7 we show that for every round r there is a round $r' > r$ in which a leader is committed. In particular, this means that for every round r with median timestamp ts, there will be eventually a committed leader with a high enough timestamp for r to be garbage collected.

Fairness. First note that since every round has at least $2f + 1$ vertices, the median timestamp of a round always belongs to an honest party. Let p_i be a party that broadcast a vertex v at some round r at time t after GST, we show that all honest parties order v. By the assumption on the reliable broadcast latency, all honest parties reliably deliver v before time $t + \Delta$. Let p_j be the first party that advances to round r. In Section 7 we show that if an honest party advances to round r at time t after GST, then all honest parties advance to round r no later than at time $t + 2\Delta$. Therefore, p_j advanced to round r not before $t - 2\Delta$. Therefore, the timestamp of round r is at least $t - 2\Delta$. Thus, round r is garbage collected only after a leader v' with timestamp higher than $t + \Delta$ is ordered. By the way the leader's timestamp is computed there is at least one vertex v'' in $v'.strongEdges$ that broadcast by an honest party after time $t + \Delta$. Therefore, by the manner weak edges are added, there is an edge between v'' and v. Fairness follows since v and v'' are in v''s casual history and thus both ordered together with v'.

7 PROOFS

In this Section we provide proofs of correctness for both versions of BullShark.

7.1 BullShark With Fallback

Total order. Note that at any given time parties might have slightly different local DAGs. This is because some vertices may be delivered at some parties but not yet at others. However, since we use reliable broadcast for each vertex v, and wait for the entire causal history of v to be added to the before we add v, we get the following important observation:

Observation 1. *For every two honest parties p_i and p_j we get:*

- *For every round r, $\bigcup_{r>0} DAG_i[r]$ is eventually equal to $\bigcup_{r>0} DAG_j[r]$.*
- *For any given time t and round r, if $v \in DAG_i[r] \wedge v' \in DAG_i[r]$ s.t. $v.source = v'.source$, then $v = v'$. Moreover, for every round $r' < r$, if $v'' \in DAG_i[r']$ and there is a path from v to v'', then $v'' \in DAG_j[r']$ and there is a path between v' to v''.*

To totally order the vertices in the DAG, each party p_i locally interprets DAG_i (there is no extra communication on top of building the DAG). To this end, p_i divides its DAG into waves of 4 rounds each. Every wave has 3 leaders that can potentially be committed: 2 steady-state leaders and one fallback leader. The steady-state leaders are two pre-defined vertices, one in the first round of the wave and the other in the third. The fallback leader is a vertex in the first round of the wave that is selected by the randomness produced in the fourth round of the wave. To make sure a fallback leader and a steady state leader are not committed in the same wave, each party can only vote for either the fallback leader or the steady-state ones. In the code, *steadyVoters[w] fallbackVoters[w]* contain all the parties that can vote for steady-state or fallback leaders in wave w, respectively. We say that a party p_i *determines* p_j *vote type* to be a steady-state (fallback) in wave w if its *steadyVoters*$[w]$

($fallbackVoters[w]$) contains p_j. Moreover, as we show in the next claim, all parties agree on p_j's vote type in wave w. This, in particular, means that Byzantine parties cannot equivocate or hide their vote (a nice property that we get from using reliable broadcast as a building block).

Claim 1. *For every party p_i and round r, each party p_j determines at most one vote type for p_i in wave w. Moreover if p_j and p_k determine vote type T and T' for p_i in wave w, respectively, then $T = T'$.*

Proof: The first part of the claim follows from the code of function *determine_party_vote_type*. This function is called by a party p_j whenever it adds a new vertex v to $DAG_j[r]$ such that r is the first round of a wave, and the source of the vertex (a party p_i) is either added to *steadyVoters[w]* or *fallbackVoters[w]*. The second part of the claim follows from Observation 1 and the fact (by the code of try_add_to_DAG) that v is added to the DAG only after all its causal history is added. This guarantees that for every wave w and party p_i try_steady_commit and try_fallback_commit are called with the same parameters and thus return the same result. This in turn guarantees that all parities that determine p_i's vote type in wave w see the same type.

There are two possible ways to commit a leader v in BullShark. The first is to *directly* commit it when either try_steady_commit or try_fallback_commit, called with v, return true. The second option is to *indirectly* commit it when it is added to *leaderStack* in Line 116 or 119. In both cases, to commit a leader in wave w, we count the number of vertices in some round (depending on the leader type and whether we directly or indirectly commit it) in w that have a strong path to the leader and their vote corresponds to the leader's type. We first show that steady state and fallback leaders cannot be directly committed in the same wave.

Claim 2. *If a party p_i directly commits a steady-state leader in wave w, then no party commits (directly or indirectly) a fallback leader in wave w, and vice versa.*

Proof: Consider a steady state leader vertex v committed by a party p_i in round r in wave w. By the code, to directly commit a leader vertex a party need to determine the vote type of at least $2f + 1$ parties in the wave to be the same as the leaders. Similarly, to indirectly commit a vertex leader, a party needs to determine the vote type of at least $f + 1$ parties in the wave to be the same as the leaders. Since p_i directly commits state leader vertex v in wave w, it determines $2f + 1$ parties as steady state voters in wave w. Since there are $3f + 1$ parties in total, by Claim 1, no other party determines more than f parties as fallback voters in wave w. Therefore, no other party commit (directly or indirectly) a fallback leader in wave w. From symmetry, the same argument works in the other direction.

For the proof of the next lemmas we say that a party p_i *consecutively directly commit* leader vertices v_i and v_i' if p_i directly commits them in rounds r_i and $r_i' > r_i$, respectively, and does not directly commit any leader vertex between r_i and r_i'. In the next claims we are going to show that honest parties commit the same leaders and in the same order:

Claim 3. *Let v_i and v_i' be two leader vertices consecutively directly committed by a party p_i in rounds r_i and $r_i' > r_i$, respectively. Let v_j and v_j' be two leader vertices consecutively directly committed by a party p_j in rounds r_j and $r_j' > r_j$, respectively. If $r_i \leq r_j \leq r_i'$, then both p_i and p_j (directly or indirectly) commit the same leader in round $min(r_i', r_j')$.*

Proof: Claim 2 implies that that there is at most one committed leader in each round. Thus, if $r_i' = r_j'$ we are done. Otherwise, assume without lost of generality that $r_i' < r_j'$. Thus, if $r_j = r_i'$ we are done. Otherwise, we need to show that p_j indirectly commits v_i' in r_i'.

By the code of commit_leader, after p_j directly commits v_j' in round r_j' it tries to indirectly commit leaders in round numbers smaller than r' until it reaches round $r_j < r_i'$. Let $r_i' < r < r_j'$, be the smallest number between r_i' and r_j' in which p_j (directly or indirectly) commits a leader v. Consider two cases:

- Vertex v_i' is a steady-state leader. Note that $r > r_i' + 1$ since only odd rounds have potential leaders. Since p_i directly commits v_i' in round r_i', there is a set C of $2f + 1$ vertices in $DAG_i[r_i' + 1]$ with strong paths to v_i' and with v_i''s types. By observation 1, Claim 1, and quorum intersection, there are at least $f + 1$ vertices in $DAG_j[r_i' + 1]$ with v_i''s vote type and strong paths from the v to them.
- Vertex v_i' is a fallback leader. By Claim 2, no leader is committed in round $r + 2$. Thus, $r > r_i' + 3$. Since p_i directly commits v_i' in round r_i' and v_i, there is a set C of $2f + 1$ vertices in $DAG_i[r_1' + 3]$ with strong paths to v_i' and with v_i''s types. By observation 1, Claim 1, and quorum intersection, there are at least $f + 1$ vertices in $DAG_j[r_i' + 3]$ with v_i''s vote type and strong paths from the v to them.

I both cases p_j counts (in *ssVotes* or *fbVotes*) at least $f + 1$ votes for the leader. In addition, by observation 1 and Claim 1, since in both cases there are at least $2f + 1$ vertices with the v's type, there are at most f vertices with the opposite type. Thus, p_j counts at most f votes for the other leader. Therefore, by Lines115-120 in commit_leader, p_j indirectly commits v_i'.

Claim 4. *Let v_i and v_i' be two leader vertices consecutively directly committed by a party p_i in rounds r_i and $r_i' > r_i$, respectively. Let v_j and v_j' be two leader vertices consecutively directly committed by a party p_j in rounds r_j and $r_j' > r_j$, respectively. Then p_i and p_j commits the same leaders between rounds $max(r_i, r_j)$ and $min(r_i', r_j')$, and in the same order.*

Proof: If $r_i' < r_j$ or $r_j' < r_i$, then we are trivially done because there are no rounds between $max(r_i, r_j)$ and $min(r_i', r_j')$. Otherwise, assume without lost of generality that $r_i \leq r_j \leq r_i'$. By Claim 3, both p_i and p_j (directly or indirectly) commit the same leader in round $min(r_i', r_j')$. Assume without lost of generality that $min(r_i', r_j') = r_i'$. Thus, by Claim 2, both p_i and p_j commit v_i' in round r_i' and v_j in round r_j. By the code of commit_leader, after (directly or indirectly) committing a leader, parties try to indirectly commit leaders in smaller round numbers until they reach a round in which they previously directly committed a leader. Therefore both p_i and p_j will try to indirectly commit all leaders going down from $r_i' =$

$min(r'_i, r'_j)$ to $r_j = max(r_i, r_j)$. Since v'_i appears in both DAG_i and DAG_j, by Observation 1, all vertices in DAG_i such that there is a path from v'_i to them appear also in DAG_j. The claim follows from the deterministic code of the function commit_leader.

By inductively applying Claim 4 for every pair of honest parties we get the following:

Corollary 7.1. *Honest parties commit the same leaders and in the same order.*

For the next lemma we say that the *causal history* of a vertex leader v in the DAG is the set of all vertices such that there is a path from v to them.

Lemma 7.2. *Algorithms 1, 2, 3, and 4 satisfy Total order.*

Proof: By Corollary 7.1, honest parties commit the same leaders and in the same order. By the code of the order_vertices procedure, parties iterate on the committed leaders according to their order and *a_deliver* all vertices in their causal history by a pre-defined deterministic rule. The lemma follows by Observation 1 since all honest parties has the same casual history in their DAG for every committed leader.

Agreement and Validity.

Lemma 7.3. *Algorithms 1, 2, 3, and 4 satisfy Agreement.*

Proof: Assume some honest party p_i outputs $a\_deliver(v_i.block, v_i.round, v_i.source)$. We will show that every honest party p_j outputs it as well. By the code of order_vertices, there is a leader vertex v that p_i committed such that v_i is in v's casual history. By Observation 1, the v's casual histories in DAG_i and DAG_j are the same. Thus, by code of order_vertices, we only need to show that p_j eventually commit leader vertex v. Let v' be the leader vertex with the lowest number that is higher than $v.round$ that p_i directly commits. Let v'' be the vertex that triggers this direct commit, i.e., the vertex v'' that passed to the try_ordering function that calls determine_party_vote_type, which in turn commits v'. By Observation 1, p_j eventually add v'' to DAG_j and call try_ordering with v''. By Observation 1 again, the casual history of v'' in DAG_i is equivalent the casual history of v'' in DAG_j. Hence, p_j directly commits v' as well. Since the casual history of v' in DAG_i is also equivalent the casual history of v' in DAG_j, p_j also commits v.

By the Liveness (Agreement and Validity) properties of reliable broadcast and since it is enough for parties to deliver $2f + 1$ vertices in a round in order to move to the next one, the DAG grows indefinitely:

Observation 2. *For every round r and honest party p_i, $DAG_i[r]$ eventually contains a vertex for every honest party.*

In the next to claims we show that for every round r there is an honest party p_i that commit a leader in a round higher than r with probability 1. First, we show that if it is not the case, then starting from some point the vote type of all parties is fallback. Note that this is true also for Byzantine parties since thanks to the reliable broadcast Byzantine parties cannot lie about their casual history.

Claim 5. *Consider an honest party p_i. If there is a wave w after which no honest party commits a leader, then in all waves $w' > w + 1$ p_i determines the vote type of all parties that reach w' in DAG_i as fallback.*

Proof: Let $w' > w + 1$ be a wave that start after w. By the claim assumption no honest party commits a leader in wave $w' - 1$. Let r be the first round of wave w'. Consider a party p_j for which p_i has a vertex v_j in $DAG_i[r]$ By the code, p_i calls try_ordering with v_j, which in turn calls determine_party_vote_type to determine p_j's vote type for w'. By Observation 1, the casual history of v_j in DAG_j is equivalent to the casual history of v_j in DAG_i. The claim follows from the code of determine_party_vote_type. Since p_j did not commit a leader in wave w', both functions try_steady_commit and try_fallback_commit return falls p_i' invocation of determine_party_vote_type. Therefore, p_i sets p_j's vote type in w' to fallback.

The following claim is a known property of all to all communication, which sometimes referred as *common core* [15]. We provide proof for completeness.

Claim 6. *For every wave w and party p_i. Let r be the first round of w. If $|DAG_i[r + k]| \geq 2f + 1, k \in \{0, 1, 2, 3\}$, then there is a set $C \subseteq DAG_i[r]$ such that $|C| = 2f + 1$ and for every vertex $v \in C$ there are $2f + 1$ vertices in $DAG_i[r + 3]$ with strong paths to v.*

Proof: The proof follows from the fact that every vertex in every round of the DAG has at least $2f + 1$ strong edges to vertices in the previous round. In particular, it is easy to show by a counting argument that there is one vertex $u \in DAG_i[r_1]$ such that $f + 1$ vertices in $DAG_i[r+2]$ has a strong edge to u. Therefore, by quorum intersection, every vertex in $DAG_i[r + 3]$ has a strong path to u. Let $C \subseteq DAG_i[r]$, $|C| = 2f + 1$ be the set of vertices that u has a strong path to, then every vertex in $DAG_i[r + 3]$ has a strong path to every vertex in C. The lemma follows since there are at least $2f + 1$ vertices in $DAG_i[r + 3]$.

Next, we use the fact that fallback leaders are hidden from adversary until the last round of a wave to prove the following:

Claim 7. *Consider a party p_i and a wave w such that p_i determines the vote type of all parties that reach w in DAG_i as fallback. Then the probability of p_i to commit the fallback vertex leader of w is at least $2/3$.*

Proof: Let r be the first round of w. By the assumption, the vote type of all parties with vertices in $DAG_i[r+3]$ is fallback. Therefore, by Claim 6, there are at least $2f + 1$ vertices in the first round of w that satisfy the fallback commit rule. That is, there is a set C of $2f+1$ parties such that if any of them is elected to be the fallback leader, then p_i will commit it. Since the fallback leader is elected with the randomness produced in round $r + 3$, the set C is determined before the adversary learns the leader. Therefore, even though the adversary fully controls delivery times, the probability for the elected leader to be in C is at least $2f + 1/3f + 1 > 2/3$.

Claim 8. *For every wave w, there is an honest party that with probability 1 commits a leader in a wave higher than w.*

Proof: Assume by a way of contradiction no honest party commits a leader in a wave higher than w. By Observation 2, for every round r and honest party p_i, $DAG_i[r]$ eventually contains at least $2f + 1$

vertices. Moreover, by Claim 5, there is an honest party p_i that determines the vote type of all parties that reach $w' > w + 1$ in DAG_i as fallback. Therefore, by Claim 7, the probability of p_i to commit the fallback leader in any wave $w' > w + 1$ is at least $\frac{2}{3}$. Hence, with probability 1, there is a wave higher than w that p_i commits.

We next use Claim 8 to prove Validity.

Lemma 7.4. *Algorithms 1, 2, 3, and 4 satisfy Validity.*

Proof: Let p_i be an honest party that calls $a\_bcast(b, r)$, we need to show that all honest parties output $a\_deliver(b, r, p_i)$ with probability 1. By the code p_i pushes b in the *blockToPropose* queue. By Observation 2, p_i advanced unbounded number of rounds and thus creates unbounded number of vertices. Therefore, eventually p_i will create a vertex v_i with b and reliably broadcast it. By the Validity property of reliably broadcast, all honest parties will eventually add it to their DAG. That is, for every honest party p_j, there is a round number r_i such that $v_i \in DAG_j[r_i]$. By the code of create_new_vertex, every vertex that p_j creates after v_i is added to $DAG_j[r_i]$ have a path to v_i (either with strong links or weak links).

Therefor, by Claim 8, there is an honest party p_j that with probability 1 commits a leader vertex with a path to v_i. Thus, by the code of order_vertices, p_i outputs $a\_deliver(b, r, p_i)$ with probability 1. Since p_i is honest, we get that by Lemma 7.3 (Agreement), all honest parties output $a\_deliver(b, r, p_i)$ with probability 1.

Integrity.

Lemma 7.5. *Algorithms 1, 2, 3, and 4 satisfy Integrity.*

Proof: An honest party p_i outputs $a\_deliver(v'.block, v'.round, v'.source)$ only if node v' is in p_i's DAG (i.e., $v' \in \bigcup_{r>0} DAG_i[r]$). Node v' is added to p_i's DAG upon the reliable broadcast $r\_deliver(v', v'.round, v'.source)$ event. Therefore, the Lemma follows from the Integrity property of reliable broadcast.

7.2 Partially Synchronous BullShark

The proof of the Integrity property is identical to the proof of Lemma 7.5. For the rest of the properties, due to similarities between the protocols and to avoid argument duplication, we will follow the structure of Section 7.1 and sometimes explain how to adapt claims' proofs.

To be consistent with the BullShark with fallback presentation, waves here are also consist of 4 rounds, each with a pre-defined leader in the first and fourth rounds (we could have waves of 2 rounds since we do not have the fallback leader).

Total order. Observation 1 applies in this case as well because the protocol to build the DAG is the same. Claim 1 trivially holds here since there is only one possible vote type and Claim 2 holds since there are no fallback leaders. The proofs of Claims 3 and 4 apply to Algorithm 5 as well. Therefore, Corollary 7.1 applies and since we use the same order_vertices procedure in both protocols we get:

Lemma 7.6. *Algorithms 1, 2, 3, and 5 satisfy Total order.*

Agreement and Validity.

The proof of the Agreement property is identical to the proof of Lemma 7.3 and Observation 2 holds since the algorithm to build the DAG is the same as in BullShark with fallback. To proof Validity for the eventually synchronous variant of BullShark we do not need Claims 5 and 8. Instead, we use the fact that GST eventually occurs. We prove the protocol under the assumption that honest parties set their timeouts to be larger than 3Δ and the following holds for the reliable broadcast building block:

Property 1. *Let t be a time after GST. If an honest party reliably broadcasts a message at time t or an honest party delivers a message at time t , then all honest parties deliver it by time $t + \Delta$.*

The above property is the equivalent to the reliable broadcast Validity and Agreement properties in the asynchronous model. To the best of our knowledge, it is satisfied by all reliable broadcast protocol since before delivering a message honest parties echo it to all other honest parties.

Claim 9. *Let w be a wave such that all honest parties advances to the first round of w after GST. Let p_1 and p_2 be their first and second pre-defined leaders of w, respectively. If p_1 and p_2 are honest, then all honest parties commit a leader in w.*

Proof: let r be the first round of w. First we show that all honest parties advance to round $r + 1$ within 2Δ time of each other. By Observation 2, all honest parties eventually advance to round $r + 1$. Let party p_i be the first honest party that advances to round $r+1$ and denote by t the time it happened. By the code of try_advance_round, $|DAG_i[r]| \geq 2f+1$. By Property 1, by time $t+\Delta$ $|DAG_j[r]| \geq 2f+1$ for all honest parties. Therefore, by Line 55, all honest party advance to round r by time $t + \Delta$. In particular, the first leader of wave w, p_1. Thus, p_1 broadcasts its vertex v_1 in round r no later than time $t + \Delta$, and by Property 1, all honest deliver it by time $t + 2\Delta$. Therefore, by Line 40 and the code of try_advance_round, all honest parties advance to round $r + 2$ by time $t + 2\Delta$.

Next we show that all honest parties advance to round $r+2$ with 3Δ time of each other. Since all honest parties advance to round $r + 1$ within 2Δ time of each other, then they start their timeouts at round $r + 1$ within 2Δ time of each other. Let party p_j be the first honest party that advances to round $r + 2$. If the first honest party waits for timeout (the if in Line 46) to advance to round $r + 2$, then all honest parties advance to round $r+2$ within 2Δ. Otherwise, p_j has $2f + 1$ vertices in $DAG_j[r + 1]$ with strong path to v'. By property 1, all other honest parties will deliver this vertices and advance to round $r + 2$ within 3Δ from p_j.

By the assumption, the second leader of the wave, p_2, is honest and will broadcast vertex v_2 in round $r + 2$ at most 3Δ after the first honest party advances to $r + 2$. Since the timeouts are larger than 4Δ, all honest will advance to round $r + 3$ within Δ of each other (by Line 42, all honest wait to deliver the leader's vertex or for a timeout). Moreover, they will all add a strong edge to v_2 in their vertex in round $r + 3$.

Since all honest advance to round $r + 3$ within Δ of each other and the timeouts are larger than 2Δ, they will all wait for each other's vertices before advancing to the next round. Therefore, all honest will get $2f + 1$ vertices in round $r + 3$ with strong paths to the second vertex leader of the wave v_2. Thus, all honest commit a leader in wave w.

The Validity property is proved under the assumption that eventually (after GST) there will be a wave in which both leaders are honest. For example, this assumption holds for every full permutation of the parties or if we maintain a fixed leader for the full wave. To avoid repetition, we omit the proof of the following lemma as it is similar to the proof of Lemma 7.4. All we need to do to adapt it is to remove all appearances of "with probability 1" and replace the reference to Claim 8 with Claim 9.

LEMMA 7.7. *Algorithms 1, 2, 3, and 5 satisfy Validity.*

8 IMPLEMENTATION

We implement a networked multi-core eventually sysnchrnous BullShark party forking the Narwhal project[3]. Narwhal provides the structured DAG used at the core of BullShark, which we modify to support fast-path in partial synchrony as described in Section 3.2. Additionally, it provides well-documented benchmarking scripts to measure performance in various conditions, and it is close to a production system (it provides real networking, cryptography, and persistent storage). It is implemented in Rust, uses `tokio`[4] for asynchronous networking, `ed25519-dalek`[5] for elliptic curve based signatures, and data-structures are persisted using `Rocksdb`[6]. It uses TCP to achieve reliable point-to-point channels, necessary to correctly implement the distributed system abstractions. By default, the Narwhal codebase runs the Tusk consensus protocol [19]; we modify the `proposer` module of the `primary` crate and the `consensus` crate to use BullShark instead. Implementing BullShark requires editing less than 200 LOC, and does not require any extra protocol message or cryptographic tool. We are open-sourcing BullShark[7] along with any Amazon web services orchestration scripts and measurements data to enable reproducible results[8].

9 EVALUATION

We evaluate the throughput and latency of our implementation of BullShark through experiments on AWS. We particularly aim to demonstrate that (i) BullShark achieves high throughput even for large committee sizes, (ii) BullShark has low latency even under high load, in the WAN, and with large committee sizes, and (iii) BullShark is robust when some parts of the system inevitably crash-fail. Note that evaluating BFT protocols in the presence of Byzantine faults is still an open research question [9].

We deploy a testbed on AWS, using `m5.8xlarge` instances across 5 different AWS regions: N. Virginia (us-east-1), N. California (us-west-1), Sydney (ap-southeast-2), Stockholm (eu-north-1), and Tokyo (ap-northeast-1). Parties are distributed across those regions as equally as possible. Each machine provides 10Gbps of bandwidth, 32 virtual CPUs (16 physical core) on a 2.5GHz, Intel Xeon Platinum 8175, 128GB memory, and runs Linux Ubuntu server 20.04. We select these machines because they provide decent performance and are in the price range of 'commodity servers'.

In the following sections, each measurement in the graphs is the average of 2 independent runs, and the error bars represent one standard deviation; errors bars are sometimes too small to be visible on the graph. Our baseline experiment parameters are 10 honest parties, a maximum block size of 500KB, and a transaction size of 512B. We instantiate one benchmark client per party (collocated on the same machine) submitting transactions at a fixed rate for a duration of 5 minutes. The leader timeout value is set to 5 seconds. When referring to *latency*, we mean the time elapsed from when the client submits the transaction to when the transaction is committed by one party. We measure it by tracking sample transactions throughout the system.

9.1 Benchmark in the common case

Figure 2 illustrates the latency and throughput of BullShark, Tusk and HotStuff for varying numbers of parties.

HotStuff The maximum throughput we observe for HotStuff is 70,000 tx/s for a committee of 10 parties, and lower (up to 50,000 tx/s) for a larger committee of 20, and even lower (around 30,000 tx/s) for a committee of 50. The experiments demonstrate that HotStuff does not scale well when increasing the committee size. However, its latency before saturation is low, at around 2 seconds.

Tusk Tusk exhibits a significantly higher throughput than HotStuff. It peaks at 110,000 tx/s for a committee of 10 and at around 160,000 tx/s for larger committees of 20 and 50 parties. It may seem counter-intuitive that the throughput increases with the committee size: this is due to the implementation of the DAG not using all resources (network, disk, CPU) optimally. Therefore, more parties lead to increased multiplexing of resource use and higher performance [19]. Despite its high throughput, Tusk's latency is higher than HotStuff, at around 3 secs (for all committee sizes).

BullShark BullShark strikes a balance between the high throughput of Tusk and the low latency of HotStuff. Its throughput is significantly higher than HotStuff, reaching 110,000 tx/s (for a committee of 10) and 130,000 tx/s (for a committee of 50); BullShark's throughput is over 2x higher than HotStuff's. Bullshark is built from the same DAG as Tusk and thus inherits its scalability allowing it to maintain high performance for large committee sizes. Contrarily to Tusk, the DAG of BullShark does not run at network speed as it sometimes wait for leaders and votes (see Section 4). This may explain the 30% throughput reduction with respect to Tusk. BullShark's selling point is however its low latency, at around 2 sec no matter the committee size. BullShark's latency is lower than Tusk since it commits within 2 DAG rounds while Tusk requires 4. BullShark's latency is comparable to HotStuff and 33% lower than Tusk. Figure 3 highlights this trade-off by showing the maximum throughput that can be achieved by HotStuff, Tusk, and Bullshark while keeping the latency under 2.5s and 5s. Tusk and Bullshark scale better than HotStuff when increasing the committee size; there is no dotted line for Tusk since it cannot commit transactions in less than 2.5s.

9.2 Benchmark under crash-faults

Figure 4 depicts the performance of HotStuff, Tusk, and BullShark when a committee of 10 parties suffers 1 to 3 crash-faults (the maximum that can be tolerated in this setting). HotStuff suffers a massive degradation in throughput as well as a dramatic increase in latency. For 3 faults, the throughput of HotStuff drops by over 10x and its latency increases by 15x compared to no faults. In contrast, both

[3] https://github.com/facebookresearch/narwhal
[4] https://tokio.rs
[5] https://github.com/dalek-cryptography/ed25519-dalek
[6] https://rocksdb.org
[7] https://github.com/asonnino/narwhal/tree/bullshark
[8] https://github.com/asonnino/narwhal/tree/bullshark/benchmark/data

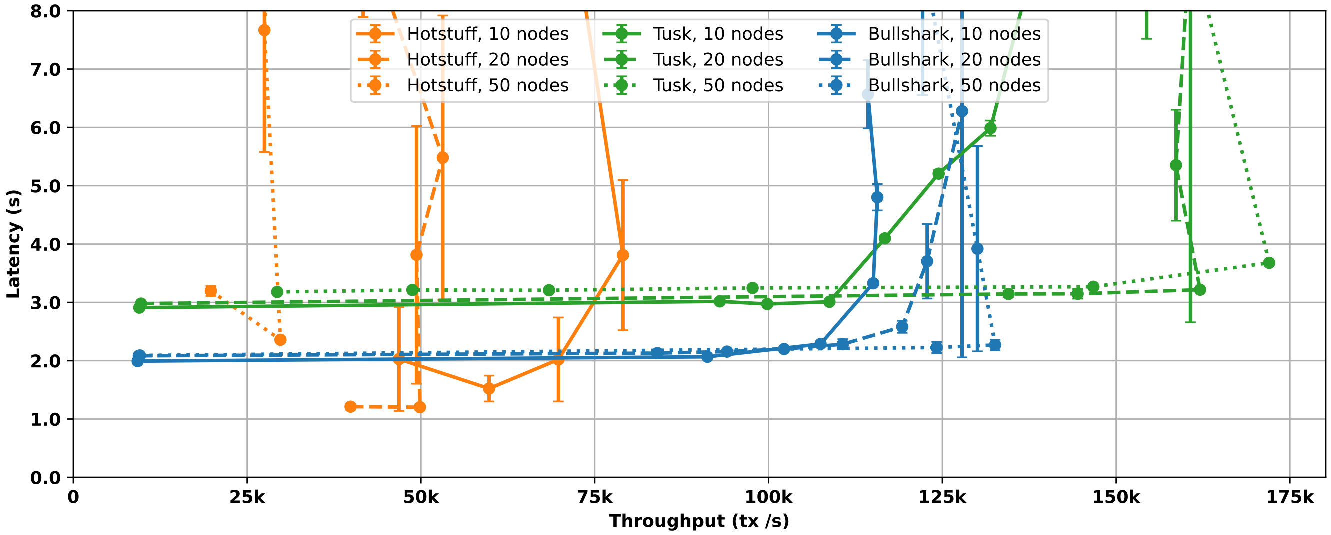

Figure 2: Comparative throughput-latency performance of HotStuff, Tusk, and BullShark. WAN measurements with 10, 20, 50 parties. No faulty parties, 500KB maximum block size and 512B transaction size.

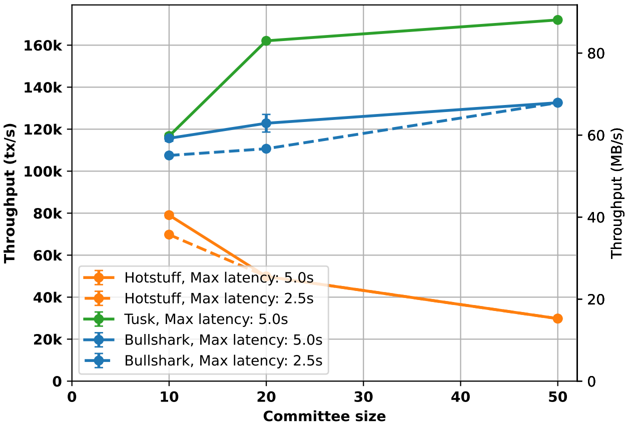

Figure 3: Maximum achievable throughput of HotStuff, Tusk, and BullShark, keeping the latency under 2.5s and 5s. WAN measurements with 10, 20, 50 parties. No faulty parties, 500KB maximum block size and 512B transaction size.

Tusk and BullShark maintain a good level of throughput: the underlying DAG continues collecting and disseminating transactions despite the crash-faults, and is not overly affected by the faulty parties. The reduction in throughput is in great part due to losing the capacity of faulty parties. When operating with 3 faults, both Tusk and BullShark provide a 10x throughput increase and about 7x latency reduction with respect to HotStuff.

9.3 Performance under asynchrony

HotStuff has no liveness guarantees when the eventual synchrony assumption does not hold (before GST), either due to (aggressive) DDoS attacks targeted against the leaders [37] or adversarial delays on the leaders' messages as experimentally proven in prior work [19, 22] . That is, the throughput of the system falls to 0. The same can happen to the partially synchronous version of BullShark.

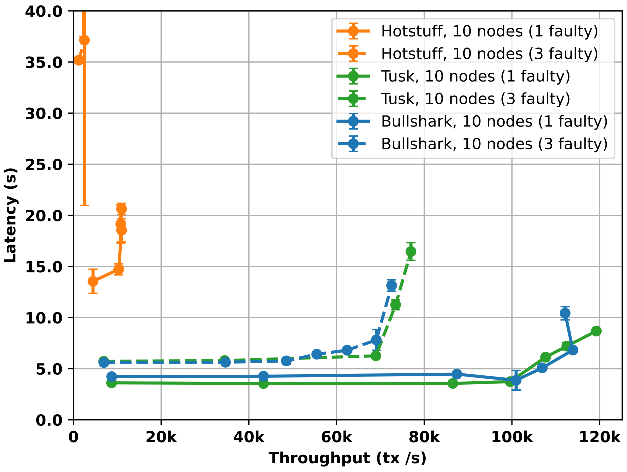

Figure 4: Comparative throughput-latency under crash-faults of HotStuff, Tusk, and BullShark. WAN measurements with 10 parties. Zero, one, and three crash-faults, 500KB maximum block size and 512B transaction size.

The reason is that whenever a party becomes the leader for some round, its proposal can be delayed such that all other parties timeout for that round. In order to avoid this attack, Tusk and DAG-Rider elects leaders unpredictably after the DAG is constructed which makes such attacks impossible. The purpose of the fallback mode of BullShark is to maintain the same liveness properties as Tusk and DAG-Rider under asynchrony without compromising on performance during periods of synchrony. If the voting type of all parties is fallback, then BullShark acts as Tusk. In the fallback mode, BullShark thus renounces to its latency advantage with respect to Tusk in order to remain live under asynchrony. As any asynchronous protocol, the performance of both Tusk and BullShark during periods of asynchrony can be arbitrarily bad as they depend on the network conditions (which guarantee delivery after unbounded time). When the period of asynchrony ends, parties change their voting type to steady-state, and BullShark offers again its state-of-the-art latency.

10 RELATED WORK

In this Section we discuss other prior works relevant to BullShark and a more in depth comparison with the systems against which we evaluate.

Performance comparisons: We compare BullShark with Tusk [19] and HotStuff [41]. Tusk is the most similar system to BullShark. It is a zero-message consensus protocol built on top of the same structured DAG as BullShark. It is however fully asynchronous while BullShark is partially-synchronous fast path. HotStuff is an established partially-synchronous protocol running at the heart of a number of projects [1–5], and a successor of the popular Tendermint [12].

We aim to compare BullShark with related systems as fairly as possible. An important reason for selecting Tusk[9] and HotStuff[10] is because they both have open-source implementations sharing deep similarities with our own. They are both written in Rust using the same network, cryptographic and storage libraries than ours. They are both designed to take full advantage of multi-core machines and to run in the WAN.

We limit our comparison to these two systems, thus omitting a number of important related works such as [12, 16, 17, 24, 26, 38, 40]. A practical comparison with those systems would hardly be fair as they do not provide an open-source implementations comparable to our own. Some selected different cryptographic libraries, use different cryptographic primitives (such as threshold signatures), or entirely emulate all cryptographic operations. A number of them are written in different programming languages, do not provide persistent storage, use a different network stack, or are not multi-threaded thus under-utilizing the AWS machines we selected. Most implementations of prior works are not designed to run in the WAN (e.g., have no synchronizer), or are internally sized to process empty transactions and are thus not adapted to the 512B transaction size we use. Instead, we provide below a discussion on the performance of alternatives based on their reported work.

Partially-synchronous protocols: Hotstuff-over-Narwhal [19] and Mir-BFT [39] are the most performant partially synchronous consensus protocols available. The performance of the former is close to BullShark under no faults given that they share the same mempool implementation. However, BullShark performs considerably better under faults and the engineering effort of Hotstuff-over-Narwhal is double that of BullShark. The extra code required to implement BullShark over Narwhal is about 200 LOC[11] (Alg. 5) whereas the extra code of Hotstuff is more than 4k LOC. Additionally, BullShark adapts to an asynchronous environment with the fallback protocol unlike Hotstuff that will completely forfeit liveness during asynchrony leading to an explosion of the confirmation latency (see Figure 4 of Section 9).

For Mir-BFT with transaction sizes of about 500B (similar to our benchmarks), the peak performance achieved on a WAN for 20 parties is around 80,000 tx/sec under 2 seconds – a performance comparable to our baseline HotStuff. Impressively, this throughput decreases only slowly for large committees up to 100 nodes (at 60,000 tx/sec). Crash-faults lead to throughput dropping to zero for up to 50 seconds, and then operation resuming after a reconfiguration to exclude faulty nodes. BullShark offers higher performance (almost 2x), at the same latency.

DAG-based protocols: The DAG have been used in the context of Blockchains in multiple systems. Hashgraph [8] embeds an asynchronous consensus mechanism into a DAG without a round-by-round step structure which results to unclear rules on when consensus is reached. This consequently results on an inability to implement garbage collection and potentially unbounded state. Finally, Hashgraph uses local coins for randomness, which can potentially lead to exponential latency.

A number of blockchain projects build consensus over a DAG under open participation, partial synchrony or asynchrony network assumptions. GHOST [35] proposes a finalization layer over a proof-of-work consensus protocol, using sub-graph structures to confirm blocks as final potentially before a judgment based on longest-chain / most-work chain fork choice rule can be made. Tusk [19] is the most similar system to BullShark. It is an asynchronous consensus using the same structured DAG as BullShark. A limitation of any reactive asynchronous protocol, such as Tusk, is that slow parties are indistinguishable from faulty ones, and as a result the protocol proceeds without them. This creates issues around fairness and incentives, since honest, but geographically distant authorities may never be able to commit transactions submitted to them. Further, Tusk relies on clients to re-submit a transaction if it is not sequenced in time, due to leaders being faulty. In contrast, both versions of BullShark satisfy fairness after GST while ensuring bounded memory via a garbage collection mechanism.

Dual-Mode Consensus Protocols: The idea of having optimistic and fallback paths in BFT consensus has first been explored by Kurasawe et al [27] with followup improvements [32, 36] on the communication complexity. However, these papers are theoretical and not designed for high-load applications hence their implementation would at best be close to the Hotstuff baseline.

The seminal work from Guerraoui et al [23] introduced Abstract, a framework in which developers can plug and play multiple consensus protocols based on the environment they plan to deploy the protocol. A followup work called the Bolt-Dumbo Transformer (BDT) [31], can be seen as instantiating of Abstract for the specific use case of a dual-mode consensus protocol. BDT takes Abstract's general proposal and instantiates it by composing three separate consensus protocols as black boxes. Every round starts with 1) a partially synchronous protocol (HotStuff), times-out the leader and runs 2) an Asynchronous Binary Agreement in order to move on and run 3) a fully asynchronous consensus protocol [24] as a fallback. Ditto [22] follows another approach that does not require these black boxes. Instead, it combines a 2-phase variant of Hotstuff with a variant of the asynchronous VABA [6] protocol for fallback. As a result it reduces the latency cost of BDT significantly, but cannot be generalized to a plug-and-play framework.

[9] https://github.com/asonnino/narwhal
[10] https://github.com/asonnino/hotstuff
[11] https://github.com/asonnino/narwhal/tree/bullshark

All the protocols above solve the problem of consensus in asynchrony, but they include the actual transactions in the proposals, hence their throughput is bounded by the one of Hotstuff. A way to increase their throughput would be to adopt the Narwhal-HS [19] approach introduced in prior work, which substitute the transaction dissemination with Narwhal as a mempool and includes only hashes of mempool batches in the proposals. This would potentially achieve similar performance to BullShark. However it would come at the steep costs of maintaining two code-bases (one for the mempool and one for the consensus), higher latency (since Narwhal does a reliable broadcast which is usually the first step of a consensus protocol) and loss of quantum-safety (since they all use threshold signatures to provide Safety with lower communication complexity). Unlike these "hybrids", BullShark provides both the theoretical contribution of being the first BAB with all the good properties we already described, the practical contribution of significant latency gains in synchrony and the usability contribution of modifying only 200 LOC from the base-protocol Tusk.

11 DISCUSSION

On the foundational level BullShark is the first DAG-based zero overhead BFT protocol that achieves the best of both worlds of partially synchronous and asynchronous protocols. It keeps all the desired properties of DAG-Rider, including optimal amortized complexity, asynchronous liveness, and post quantum security, while also allowing a fast-path during periods of synchrony. BullShark's parties switch their voting type to fallback after every unsuccessful wave. An interesting future direction is to add an adaptive mechanism for parties to learn when is best to switch between the types. Interestingly, since the DAG provides full information, this mechanism can be also implemented without extra communication.

The partially synchronous version of BullShark is extremely simple (200 LOC) and highly efficient. In particular, it does not need any view-change or view-synchronization mechanisms since the DAG already encodes all the required information. When implemented over the Narwhal mempool it has $2x$ the throughput of the partially synchronous HotStuff protocol and 33% lower latency than the asynchronous Tusk protocol over Narwhal.

ACKNOWLEDGEMENTS

This work has been initiated when all authors were working at Novi at Facebook. We thank George Danezis for the insightful discussions, and the anonymous CCS reviewers for their constructive feedback.

REFERENCES

[1] 2022. Celo. https://celo.org. (2022).

[2] 2022. Cypherium. https://www.cypherium.io. (2022).

[3] 2022. Diem. https://www.diem.com. (2022).

[4] 2022. Flow. https://www.onflow.org. (2022).

[5] 2022. Thunder. https://www.thundercore.com/. (2022).

[6] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 337–346.

[7] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.

[8] Leemon Baird. 2016. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep* (2016).

[9] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, and Dahlia Malkhi. 2021. Twins: BFT Systems Made Robust. In *Principles of Distributed Systems*.

[10] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptology and information security*. Springer, 514–532.

[11] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143.

[12] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains*. Ph.D. Dissertation.

[13] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18, 3 (2005), 219–246.

[14] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE, 191–201.

[15] Ran Canetti. 1996. *Studies in secure multiparty computation and applications*. Ph.D. Dissertation. Citeseer.

[16] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[17] Benjamin Y Chan and Elaine Shi. 2020. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. 1–11.

[18] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 43, 2 (1996), 225–267.

[19] 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.

[20] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

[21] Adam Gkagol, Damian Leśniak, Damian Straszak, and Michał Świketek. 2019. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 214–228.

[22] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2021. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. *arXiv preprint arXiv:2106.10362* (2021).

[23] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT protocols. In *Proceedings of the 5th European conference on Computer systems*. 363–376.

[24] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 803–818.

[25] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. *arXiv preprint arXiv:2102.08325* (2021).

[26] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th {usenix} security symposium ({usenix} security 16)*. 279–296.

[27] Klaus Kursawe and Victor Shoup. 2005. Optimistic asynchronous atomic broadcast. In *International Colloquium on Automata, Languages, and Programming (ICALP)*. Springer, 204–215.

[28] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. 2004. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Trans. Comput.* 53, 7 (2004), 815–828.

[29] Benoît Libert, Marc Joye, and Moti Yung. 2016. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theoretical Computer Science* 645 (2016), 1–24.

[30] Julian Loss and Tal Moran. 2018. Combining Asynchronous and Synchronous Byzantine Agreement: The Best of Both Worlds. *IACR Cryptol. ePrint Arch.* 2018 (2018), 235.

[31] Yuan Lu, Zhenliang Lu, and Qiang Tang. 2021. Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As Pipelined BFT. *arXiv preprint arXiv:2103.09425* (2021).

[32] HariGovind V Ramasamy and Christian Cachin. 2005. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *International Conference On Principles Of Distributed Systems (OPODIS)*. Springer, 88–102.

[33] Maria A Schett and George Danezis. 2021. Embedding a Deterministic BFT Protocol in a Block DAG. *arXiv preprint arXiv:2102.09594* (2021).

[34] Victor Shoup. 2000. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 207–220.

[35] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*. Springer, 507–527.

[36] Alexander Spiegelman. 2021. In Search for an Optimal Authenticated Byzantine Agreement. In *35th International Symposium on Distributed Computing (DISC)*.

[37] Alexander Spiegelman, Arik Rinberg, and Dahlia Malkhi. 2021. ACE: Abstract Consensus Encapsulation for Liveness Boosting of State Machine Replication. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[38] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. 2019. Mir-BFT: High-Throughput Robust BFT for Decentralized Networks. *arXiv preprint arXiv:1906.05552* (2019).

[39] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. *CoRR* abs/1906.05552 (2019). arXiv:1906.05552 http://arxiv.org/abs/1906.05552

[40] Lei Yang, Vivek Bagaria, Gerui Wang, Mohammad Alizadeh, David Tse, Giulia Fanti, and Pramod Viswanath. 2019. Prism: Scaling bitcoin by 10,000 x. *arXiv preprint arXiv:1909.11261* (2019).

[41] 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*. 347–356.