Bullshark: DAG BFT Protocols Made Practical

Neil Giridharan
giridhn@berkeley.edu
UC Berkeley

Lefteris Kokoris-Kogias
giridhn@berkeley.edu
IST Austria

Alberto Sonnino
asonnino@fb.com
Facebook

Alexander Spiegelman
sashaspiegelman@fb.com
Facebook

ABSTRACT

We present BullShark, the first directed acyclic graph (DAG) based Byzantine Fault tolerant (BFT) protocol that is optimized for partial synchrony. BullShark inherits all the desired properties of its predecessor (DAG-Rider) such as optimal amortized complexity, asynchronous liveness, zero-overhead, and post-quantum safety, but at same time BullShark provides a practical low latency fast-path that exploits synchronous periods. In addition, we introduce a stand alone partially synchronous version of BullShark and evaluate it against the state of the art. The resulting protocol is embarrassingly simple (200 LOC on top of a DAG-based mempool implementation) and highly efficient, achieving for example, 125,000 transaction per second and 2 seconds latency with 50 nodes.

ACM Reference Format:
Neil Giridharan, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 12 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Ordering commands 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, 15, 18, 20, 25] proposed an elegant way to separate between the distribution of commands and the logic required to safely order them. The idea is simple. To propose commands, parties send them in a way that forms a casual order among them. That is, messages contain blocks of commands 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 interpret their view of it without sending any extra messages.

The pioneering work of Hashgraph [8] constructed an unstructured DAG, where each message refers to 2 previous ones, and used hashes of messages as local coin flips to totally order the DAG in asynchronous settings. Aleph [18] 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 [20], which build on previous ideas to also achieve optimal amortized complexity, post quantum security, and some notion of fairness. That is, DAG-Rider is an asynchronous Byzantine atomic broadcast (BAB), which besides totally ordering messages (commands) also guarantees that every command proposed by an honest party is eventually delivered (ordered). Moreover, by using the DAG to abstract away the communication layer, the entire ordering logic of DAG-rider spans over less than 30 lines of pseudocode.

However, although DAG-based protocols have a solid theoretical foundation, they have multiple gaps before being realistically deployable. First, they all optimize for the worst case asynchronous network assumptions and do not take advantage of synchronous periods, resulting to higher latency than existing consensus protocols [11, 31]. Second, they assume some impractical assumptions such as unbounded memory in order to preserve fairness. The only existing solution to this comes from Narwhal [15] which uses a garbage collection mechanism but does not allow for fairness even when the network is good.

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 to performance bottlenecks and relatively low throughput as shown by Narwhal [15].

In this paper we optimize the mempool co-design mindset of DAG's to the common case communication setting. First, we propose BullShark that preserves all the nice properties of DAG-Rider, and in addition, introduces a fast path that exploits common case synchronous conditions. That is, BullShark is the first BAB protocol with optimal amortized communication complexity and post quantum security that on the one hand provides 6 round-trip latency in expectation in asynchronous executions between commits (as DAG-Rider), and on the other hand provides only 2 round-trip latency during synchrony or a 50% improvement compared to 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 embedded into the 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 3 second latency with 50 correct nodes.

Conference'17, July 2017, Washington, DC, USA

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

In addition, we overcome existing practical limitations of DAG-based protocols that do not allow for fairness and garbage collection and demonstrate the power of BullShark through extensive implementation and evaluation.

1.1 Technical challenges.

In order to design and implement BullShark we had to solve foundational 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 our fast path or 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 pre-define steady state leaders. To solve this issue we needed to deconstruct the way partial synchronous protocols are designed and embed the 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 or a timeout to advance rounds.

Another challenge was to take advantage of a common-case synchronous network without sacrificing latency in the worst case To this end, we enhance BullShark to operate in two modes, *steady state* and an asynchronous *fallback*. Similarly to DAG-Rider [15], BullShark is a wave-by-wave protocol and to achieve similar latency to DAG-Rider in the worst case, every fallback wave in BullShark consists of 4 rounds. However, to optimize for the fast path latency, every steady-state wave in BullShark consists only from 2 round. Aligning the fast-path 2 round waves with the fallback 4 rounds waves seamlessly is a subtle task. Each wave has its own leader (pre-defined for steady-state waves and unpredictably elected for fallback) and parties need to track other parties modes while interpreting the DAG and make sure they do not commit conflicting leaders.

Last but not the least, to evaluate BullShark we had to resolve some practical challenges. First, all previous theoretical solutions require an 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 [15] implemented a scalable DAG and dealt exactly with these problems. Their main motivation was to use DAG as an efficient command mempool for external byzantine protocols (e.g., Hotstuff) to use. We, therefore, started from Narwhal's open source code base and adopt their garbage collection mechanism and efficient consistent broadcast. Unfortunately, however, the narwhal garbage collection mechanism directly conflicts with BullShark mechanism to provide fairness. In fact, providing 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 fairness only during synchronous periods in which case all messages by honest parties are able to get into the DAG and be ordered before the garbage collection.

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.

For the description of the protocol we assume that links between correct parties are reliable. That is, all messages among honest parties eventually arrive. Moreover, for simplicity, we assume that recipients can verify the senders identities [1]. 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*: synchrnous exections 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 process p_k can broadcast messages by calling $r\_bcast_k(m, r)$, where m is a message, $r \in \mathbb{N}$ is a round number. Every process 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 process that called the corresponding $r\_bcast_k(m, r)$. The reliable broadcast abstraction guarantees the following properties:

Agreement If a correct processes p_i outputs $r\_deliver_i(m, r, p_k)$, then every other correct process p_j eventually outputs $r\_deliver_j(m, r, p_k)$.

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

Validity If a correct process p_k calls $r\_bcast_k(m, r)$, then every correct processes p_i eventually outputs $r\_deliver_i(m, r, k)$.

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

[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.

Agreement If two correct processes 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 $f + 1$ processes call $choose\_leader(w)$, then every $choose\_leader(w)$ call eventually returns.

Unpredictability As long as less than $f + 1$ processes 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 process p_j of the call $choose\_leader(w)$ is $pr \leq \Pr[X_w = p_j] + \epsilon$.

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

Implementation examples that use PKI and a threshold signature scheme [10, 23, 26] can be found in [12, 24]. See DAG-Rider for more details on a coin implementation can be integrated into the DAG construction. It is important to note that the above implementations satisfy Agreement, Termination, and Fairness with information theoretical guarantees. That is, an assumption of computationally bounded adversary is required only for the unpredictability property. Therefore, since BullShark relies on the unpredictability property of the coin only for liveness, its safety properties are post-quantum secure.

2.3 Problem Definition

Following DAG-Rider [20], 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 $a\_deliver(m, r, k)$, 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 process. For simplicity of presentation, we assume that each process broadcasts infinitely many messages with consecutive sequence numbers.

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

Total order If a correct process p_i outputs $a\_deliver_i(m, r, k)$ before $a\_deliver_i(m', r', k')$, then no correct process p_j outputs $a\_deliver_j(m', r', k')$ without first outputting $a\_deliver_j(m, r, k)$.

Note that the above definition is agnostic to the network assumptions. However, in asynchronous executions, due to the FLP result [17], 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 4.3, 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 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 DAG in DAG-Rider [20]. 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 need from DAG-Rider in Section 3.1 and describe our DAG construction in Section 3.2.

3.1 Background

We use a directed acyclic graph (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 and it contains, 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 the DAG during different times of the execution (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 $DAG_i[]$. Vertexes are created via the $create\_new\_vertex(r)$ procedure. Each vertex in the DAG is associated with a unique round r number and a the party who generated and reliably broadcast it (the source). In addition, each vertex contains a block of transactions that were previously a_bcast by the upper BAB protocol 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. As explained later, the BAB protocol combines two modes, steady state and fallback, each of which operates in a wave by wave manner. The steady state guarantees fast deterministic progress when the network is synchronous, whereas

Algorithm 1 Data structures and basic utilities for process 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

1: **procedure** *path*(v, u) ▷ Check if exists a path consisting of strong and weak edges in the DAG
2: **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 $\forall i \in [2..k]$: $v_i \in \bigcup_{r \geq 1} DAG_i[r] \wedge (v_i \in v_{i-1}.weakEdges \cup v_{i-1}.strongEdges)$

3: **procedure** *strong_path*(v, u) ▷ Check if exists a path consisting of only strong edges in the DAG
4: **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 $\forall i \in [2..k]$: $v_i \in \bigcup_{r \geq 1} DAG_i[r] \wedge v_i \in v_{i-1}.strongEdges$

5: **procedure** *create_new_vertex*(round)
6: **wait until** $\neg blocksToPropose$.empty()
7: $v.round \leftarrow round$
8: $v.source \leftarrow p_i$
9: $v.block \leftarrow blocksToPropose$.dequeue()
10: $v.strongEdges \leftarrow DAG[round - 1]$
11: *set_weak_edges*$(v, round)$
12: **return** v

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

18: **procedure** *get_fbWave_vertex_leader*(w)
19: $p \leftarrow choose\_leader_i(w)$
20: **return** $get\_vertex(p, 4w - 3)$

21: **procedure** *get_ssWave_vertex_leader*(w)
22: $p \leftarrow get\_predefined\_leader(w)$
23: **return** $get\_vertex(p, 2w - 1)$

24: **procedure** *get_vertex*(p,r)
25: **if** $\exists v \in DAG[r]$ s.t. $v.source = p$ **then**
26: **return** v
27: **return** $\perp$

the fallback guarantees safe probabilistic progress when the network is asynchronous (under attack).

The procedure $get\_fbWave\_vertix\_leader$ gets a fallback wave number, computes the randomly elected leader of the wave and then returns the vertex the elected leader broadcast in the first round of the wave, if it is included in the DAG. Otherwise, returns $\perp$. The procedure $get\_ssWave\_vertix\_leader$ is similar but it gets a steady state wave number and considers the pre-defined leader of the wave. We assume a pre-defined and known to all parties mapping $get\_predefined\_leader$ from steady state waves to leaders.

3.2 Our DAG protocol

A detailed pseudocode is given in Algorithm 2. 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 the vertex validity. That is, (1) the source and round must match the reliable broadcast instance to prevent equivocation, (2) the vertex must has at least $2f + 1$ strong edges, (3) one of which points to a vertex created by the same source (the purpose will became clear shortly). Then, the 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 are references are already delivered. If this is not yet the case, the vertex is added to a *buffer* for a later retry. Otherwise, p_i calls the procedure $update\_validator\_mode$, which triggered the higher level BAB protocol and then tries to add all vertexes in the buffer again.

As we explain in details in the next section our BAB protocol distinguishes between steady state and asynchronous fallback modes. Both of them proceed in waves. Each steady state wave consist of two consecutive rounds and has a pre-defined leader whereas each fallback wave consists of four rounds and elect an unpredictable leader in retrospect. The DAG-Rider protocol considers only the asynchronous case and thus advances rounds as soon as the DAG contains $2f + 1$ vertexes in the current round. We, in contrast, optimize for the common case conditions and thus have to make sure that we do not advance rounds too fast. Otherwise, since the adversary can control which $2f + 1$ vertexes parties deliver first even after GST, it can prevent honest parties from making progress in the steady state mode even in executions with common case conditions.

More specifically, as we shortly explain, the first round of a steady state wave allows the pre-defined leader of the wave to "propose" and the second round allows the other parties to "vote". Therefore, although in both cases p_i will advance its round only after adding at least $2f + 1$ vertexes to its DAG in its current round (as in DAG-Rider), the logic to advance rounds in our DAG differs between odd and even rounds. In odd rounds p_i waits either for a timeout or to add a vertex created by the waves pre-defined leader. Otherwise, the adversary could make p_i advance odd rounds before delivering the wave's leader vertex even in common case conditions. In even rounds p_i waits either for a timeout or to add $2f + 1$ vertexes in the steady state mode in the current wave, which tracked in the higher level BAB protocol via the set *ssValidatorsSets*. Otherwise, the adversary could make p_i advance even rounds whithout satisfying the commit rule (see next section) even in common case conditions. The timeout is reset when p_i advances a round, commencing it by creating and broadcasting a new vertex. The new vertex, among other things, set its strong edges to point to the $2f + 1$ vertexes from the previous round in the DAG.

Algorithm 2 DAG construction, psuedocode for process p_i

Local variables:
$r \leftarrow 0$
$buffer \leftarrow \{\}$
$waitForTimer \leftarrow true$

28: **upon** $r\_deliver_i(v, r, p)$ **do**
29: **if** $v.source \neq p \wedge v.round \neq r \wedge |v.strongEdges| \geq 2f + 1 \wedge \exists v' \in v.strongEdges : v.source = v'.source$ **then**
30: **if** $\neg try\_add\_to\_dag(v)$ **then**
31: $buffer \leftarrow buffer \cup \{v\}$
32: **else**
33: **for** $v \in buffer : v.round \leq r$ **do**
34: $try\_add\_to\_dag(v)$
35: $ssWave \leftarrow \lceil r/2 \rceil$
36: **if** $r \bmod 2 == 1 \wedge (\neg waitForTimer \vee \exists v \in DAG[r] : v.source = get\_steady\_leader(ssWave))$ **then**
37: $try\_advance\_round()$
38: **if** $r \bmod 2 == 0 \wedge (\neg waitForTimer \vee \exists U \subseteq DAG[r] : |U| = 2f + 1$ and $\forall u \in U, u.source \in ssValidatorsSets[ssWave])$ **then**
39: $try\_advance\_round()$

40: **upon** timeout **do**
41: $waitForTimer \leftarrow false$
42: $try\_advance\_round()$

43: **procedure** try_add_to_dag(v)
44: **if** $\forall v' \in v.strongEdges \cup v.weakEdges : v' \in \bigcup_{k \geq 1} DAG[k]$ **then**
45: $DAG[v.round] \leftarrow DAG[v.round] \cup \{v\}$
46: $buffer \leftarrow buffer \setminus \{v\}$
47: $try\_ordering(v)$
48: **return** true
49: **return** false

50: **procedure** try_advance_round()
51: **if** $|DAG[r]| \geq 2f + 1$ **then**
52: $v \leftarrow create\_new\_vertex(r)$
53: $try\_add\_to\_dag(v)$
54: $r\_bcast_i(v, r)$
55: $waitForTimer \leftarrow true$
56: *start timer*
57: $r \leftarrow r + 1$

4 THE BULLSHARK PROTOCOL

In this section we describe BullShark, which is a DAG-based BAB protocol that optimizes latency in the common case conditions, and at the same time preserves DAG-Rider's latency in worst case condition, as well as all its other nice properties (e.g., optimal amortize complexity and post quantum safety). Similarly to DAG-Rider, BullShark requires no communication. Instead, each party observes it local copy of the DAG and totally order its nodes. In order to optimize for the common case condition while guaranteeing liveness under worst case asynchronous conditions, BullShark combines two operational modes, steady state and fallback, each of which requires different number of DAG rounds to commit and has a different mechanism to elect leaders. The main challenge, therefore, in deigning BullShark is the interplay between the modes. We divide the protocol description into two parts, were we first describe how parties keep track of other parties modes, and then we explain how parties use this information to safely commit prefixes of the DAG. In Section 5 we preset an eventually synchronous version of BullShark, which operates only steady state modes and rely on eventual synchrony for progress.

4.1 Keeping track of parties modes

Algorithm 3 BullShark part 1: protocol to update validators mode for party p_i.

Local variables:
$ssValidatorsSets[1], \leftarrow \Pi$; $fbValidatorsSets[1] \leftarrow \{\}$
For every $j > 1$ $fbValidatorsSets[j]$, $ssValidatorsSets[j] \leftarrow \{\}$

58: **upon** $a\_bcast_i(b, r)$ **do**
59: $blocksToPropose$.enqueue(b)

60: **procedure** try_ordering(v)
61: $update\_validator\_mode(v)$

62: **procedure** update_validator_mode(v)
63: $ssWave \leftarrow \lceil v.round/2 \rceil$
64: $fbWave \leftarrow \lceil v.round/4 \rceil$
65: **if** $v.source \in ssValidatorsSets[ssWave] \cup fbValidatorsSets[fbWave]$ **then**
66: **return** ▷ the mode for v's wave is already determined
67: $potentialVotes \leftarrow v.strongEdges$ ▷ v.round is a first round of a wave
68: **if** $v.source \in ssValidatorsSets[ssWave - 1] \wedge try\_steady\_commit(potentialVotes, ssWave - 1)$ **then**
69: $ssValidatorsSets[ssWave] \leftarrow ssValidatorsSets[ssWave] \cup \{v.source\}$
70: **else if** $v.source \in fbValidatorsSets[fbWave - 1] \wedge try\_falback\_commit(potentialVotes, fbWave - 1)$ **then**
71: $ssValidatorsSets[ssWave] \leftarrow ssValidatorsSets[ssWave] \cup \{v.source\}$
72: **else**
73: $fbValidatorsSets[fbWave] \leftarrow fbValidatorsSets[fbWave] \cup \{v.source\}$

74: **procedure** try_steady_commit($potentialVotes, w$)
75: $v \leftarrow get\_steady\_leader(w)$
76: **if** $|\{v' \in potentialVotes : v'.source \in ssValidatorsSets[w] \wedge strong\_path(v', v)\}| \geq 2f + 1$ **then**
77: $commit\_leader(v)$
78: **return** true
79: **return** false

80: **procedure** try_fallback_commit($potentialVotes, w$)
81: $v \leftarrow get\_fallback\_leader(w)$
82: **if** $|\{v' \in potentialVotes : v'.source \in fbValidatorsSets[w] \wedge strong\_path(v', v)\}| \geq 2f + 1$ **then**
83: $commit\_leader(v)$
84: **return** true
85: **return** false

Similarly to DAG-Rider, to interpret the DAG, each party p_i divides its local view of the DAG, DAG_i, into waves. In DAG-Rider, each wave consists of 4 rounds to guarantee progress in $O(1)$ waves in expectations. We adopt this approach to our fallback waves, but to reduce latency in the common case our steady-state waves consist of only 2 rounds.

The pseudocode appears in Algorithm 3. Each party p_i maintains the sets *ssValidatorsSets[w]* and *fbValidatorsSets[w]* to keep track of validators modes as they advance waves. In a nutshell, a party is in the steady-state mode in wave w if it committed in wave $w-1$, and otherwise it is in the fallback. Importantly, a party cannot be in both modes in the same wave. To determine parties modes, party p_i calls $update\_validator\_mode(v)$ when it adds node v to its DAG. By the code, p_i adds v only after all the nodes of $v.source$ in previous rounds are already in the DAG. If $v.round$ is not the first round of a wave, v's mode was already previously determined as does not change inside a wave. For a node v in the first round of a wave w, p_i looks into $v.strongEdges$ and determines the *votes* that v observed when deciding whether to commit one of the leaders of wave $w-1$, hence determining $v's$ mode for w.

Again, by the code, the mode of all the nodes in $votes$ as well as the mode of v in $w-1$ are already determined before $update\_validator\_mode(v)$ is called. The commit rule for both modes is $2f+1$ votes. Therefore, all is left for p_i is to count the number of nodes in $votes$ with the same mode as v in $w-1$ and a strong path to the mode's leader of $w-1$. For example, if the mode of v in $w-1$ is steady-state (fallback) and *votes* has $2f+1$ strong paths to the steady state (fallback) leader of $w-1$, then v have committed the steady-state (fallback) leader of $w-1$ and p_i sets the mode of v to be steady-state for wave w. Otherwise, p_i sets v's mode in wave w to fallback.

The goal of BullShark is to commit as fast as possible, taking advantage of periods of network synchrony. Therefore, all the parties start in the steady-state mode and stay there as long as they keep committing waves. Once a party fails to commit, it moves to the fallback mode. When enough honest parties move to the fallback mode, they are able to commit in the fallback and move back to the steady-state mode. Note that the DAG structure contains all the information in the system and thanks to the reliable broadcast properties even Byzantine nodes cannot lie about their modes. However, an asynchronous adversary can make some honest parties stay in the fallback mode forever without committing leaders by helping other honest parties commit in the steady-state mode. This is because the commit rule requires $2f+1$ nodes in some mode to commit a leader. So potentially $f+1$ honest parties plus f Byzantine can keep committing leaders in the steady state mode while f honest parties keep failing to commit in the fallback. To remedy this issue and make sure all honest parties advance and totally order all DAG nodes, p_i commits not only the leader that its own commit rule satisfied, but any leader committed by any party while determining the modes of other parties. This is done by calling the *commit_leader* procedure, which is described next.

4.2 Committing DAG prefixes

So far we described the wave commit rules and how we use them to determine parties modes. Now 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 node v of some wave by either directly its own commit rule or indirectly by learning that some other party committed it, it calls $commit\_leader(v)$. Similarly to DAG-Rider, p_i goes back wave-by-wave until the wave of the last leader (tracked in *decidedWave*) it ordered and check whether somebody might have committed and ordered their leaders. In which case, p_i should order these leaders before v. In contrast to DAG-Rider, BullShark has two potential leaders (steady-state and fallback) in every wave and we have to make sure parties commit the same leaders.

Algorithm 4 BullShark part 2: the commit protocol for party p_i

```
    Local variables:
        decidedWave ← 0
        decidedVertices ← {}
        leaderStack ← initialize empty stack

86: procedure commit_leader(v)
87:     leadersStack.push(v)
88:     ssWave ← ⌈v.round/2⌉
89:     for wave w from ssWave − 1 down to decidedWave + 1 do
90:         potentialVotes ← v'.strongEdges s.t. v' ∈ DAG[2w + 1]
                                                ∧v'.source = v.source
91:         v_s ← get_steady_leader(w)
92:         ssVotes ← {v' ∈ potentialVotes : v'.source ∈
                        ssValidatorsSets[w] ∧ strong_path(v', v_s)}
93:         if w mod 2 = 0 then
94:             v_f ← get_fallback_leader(w/2)
95:             fbVotes ← {v' ∈ potentialVotes : v'.source ∈
                            fbValidatorsSets[w/2] ∧ strong_path(v', v_f)}
96:         else
97:             fbVotes ← {}
98:         if |ssVote| ≥ f + 1 ∧ |fbVote| < f + 1 then
99:             leadersStack.push(v_s)
100:            v ← v_s
101:        if |ssVote| < f + 1 ∧ |fbVote| ≥ f + 1 then
102:            leadersStack.push(v_f)
103:            v ← v_f
104:    decidedWave ← ssWave
105:    order_vertices()

106: procedure order_vertices()
107:    while ¬leadersStack.isEmpty() do
108:        v ← leadersStack.pop()
109:        verticesToDeliver ← {v' ∈ ⋃_{r>0} DAG_i[r] | path(v, v') ∧ v' ∉
                                deliveredVertices}
110:        for every v' ∈ verticesToDeliver in some deterministic order do
111:            output a_deliver_i(v'.block, v'.round, v'.source)
112:            deliveredVertices ← deliveredVertices ∪ {v'}
```

Note that by quorum intersection and the non-equivocation property of the DAG, if some party commits a leader in either of the modes by seeing $2f+1$ votes, then all other parties see at least $f+1$ of these votes. Moreover, since a party cannot be at both modes in the same wave, if p_i sees $f+1$ votes for the leader of one of the modes ($f+1$ nodes in the leader's mode with strong paths to the leader), then no party could have committed the leader of the other mode since in this case there are at most $2f$ nodes in that mode (Byzantine parties cannot lie about their mode).

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 the leaders of a previous yet undecided wave w in the for loop in procedure *commit_leader*. Note that for the steady-state case round

$2w$ is the voting round of wave w and each node in round $2w+1$ refers to $2f+1$ potential votes via its strong edges. Since the code guarantees that each party that has v in its DAG also has a node v' in round $2w+1$ with the same source as v, parties use the strong edges of v' as the potential votes. To compute the number of votes for the steady-state leader in wave w, p_i counts how many nodes out of the potential votes are in the steady-state mode in wave w and has a strong path to the steady-state leader. As for the fallback votes, note that the for loop iterates over the steady-state waves, and since each fallback wave (4 rounds) spans over 2 steady-state waves, we compute the fallback votes only if w is even. In which case, p_i counts how many nodes out of the potential votes are in the fallback mode in wave w and has a strong path to the fallback leader.

After computing the number of steady-state and fallback votes, p_i check if one of the leaders could be committed. That is, if one of the leaders u has at least $f+1$ votes while the other has at most f. In which case p_i orders u by pushing it to the leader's stack *leaderStack* and continues to the next loop iteration to check if there are leaders to order before u. Otherwise, p_i skips the leaders of wave w and continues to the next loop iteration with v.

As we prove in Appendix A, all honest parties order the same leaders and in the same order. All is left is to apply some deterministic rule to order their causal histories. Therefore, at the end of the loop party p_i calls $order\_vertex()$, which pops all leaders from the leader's stack, and for each of them delivers, by some deterministic order, all the nodes in it causal history (strong and weak edges) that have not been delivered yet.

4.3 Garbage collection in BullShark

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

Similarly to BullShark, DAG-Rider[20], Aleph [18], and Narwhal [15] build a round-based structured DAG. Narwhal proposes a garbage collection mechanism through using the consensus decisions. However their protocol breaks the the validity property of the BAB problem, that is, 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[20] and BullShark, 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. The result is a direct observation of FLP [17] since we need to decide that a party crashed in some older round and did not produce a block, which then allows us to garbage collect the round. Therefore, in the BullShark implementation we propose a practical alternative. We maintain bounded memory but provide fairness only after GST.

The garbage collection mechanism works as follows: First we add a local time in every physical block. When a leader is committed the median timestamp of its parents is used as the time of the parent's round. From this time we substract 3Δ and look for a round with a median timestamp earlier than 3Δ from the parents' round timestamp and garbage collect from this round and before.

The intuition on why this allows for fairness after GST is simple. First, the timestamp mechanism is inspired by classic clock synchronization mechanisms and guarantees that the round time will be within Δ of the time the slowest node enters the round. Once that slow node start the round it takes 2Δ to reliable broadcast the block. Hence after 3Δ all honest node should be able to add a weak link on that block. As a result any future round (after GST) will either include the block or we can use the mechanims as a failure detector and safely agree that a party did not send a block for that round, allowing for fair garbage collection.

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 the next section, very efficient. In a nutshell, in the eventually synchronous version parties never fallback. Instead they keep interpreting the DAG waves modes as steady state. The safety argument is similar to the Asynchronous Bullshark and Termination is guaranteed after GST since all parties get honest leaders' messaged and votes before they timeout. The pseudocode, which overwrites the $try\_ordering$ procedure, appears in Algorithm 5. Note that Algorithm 5 calls some procedures from previous Algorithms.

Algorithm 5 Eventually synchronous BullShark: protocol for party p_i.

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

113: procedure try_ordering(v)
114:     if v.round mod 2 == 0 then
115:         return                                   ▷ this is a voting round
116:     ssWave ← ⌈v.round/2⌉
117:     try_commit(v.strongEdges, ssWave − 1)

118: procedure try_commit(votes, w)
119:     v ← get_steady_leader(w)
120:     if |{v' ∈ votes : strong_path(v', v)}| ≥ 2f + 1 then
121:         commit_leader(v)
122:         return true
123:     return false

124: procedure commit_leader(v)
125:     leadersStack.push(v)
126:     ssWave ← ⌈v.round/2⌉
127:     for wave w from ssWave − 1 down to decidedWave + 1 do
128:         v_s ← get_steady_leader(w)
129:         if strong_path(v, v_s) then
130:             leadersStack.push(v_s)
131:             v ← v_s
132:     decidedWave ← ssWave
133:     order_vertices()                            ▷ see Algorithm 4
```

6 IMPLEMENTATION

We implement a networked multi-core eventually sysnchrnous BullShark validator forking the Narwhal project[3]. Narwhal provides the structured DAG used at the core of BullShark. 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 [16]; 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].

7 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, (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). Validators 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. Our baseline experiment parameters are 10 honest validators, a maximum block size of 500KB, a transaction size of 512B, and one benchmark client per validator (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 validator. We measure it by tracking sample transactions throughout the system.

[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

7.1 Benchmark in the common case

Figure 1 illustrates the latency and throughput of BullShark, Tusk and HotStuff for varying numbers of validators.

HotStuff The maximum throughput we observe for HotStuff is 70,000 tx/s for a committee of 10 nodes, 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 nodes. 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 validators lead to increased multiplexing of resource use and higher performance [16]. 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. **??** 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.

7.2 Benchmark under crash-faults

Figure 3 depicts the performance of HotStuff, Tusk, and BullShark when a committee of 10 validators 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 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 validators. The reduction in throughput is in great part due to losing the capacity of faulty validators. When operating with 3 faults, both Tusk and BullShark provide a 10x throughput increase and about 7x latency reduction with respect to HotStuff.

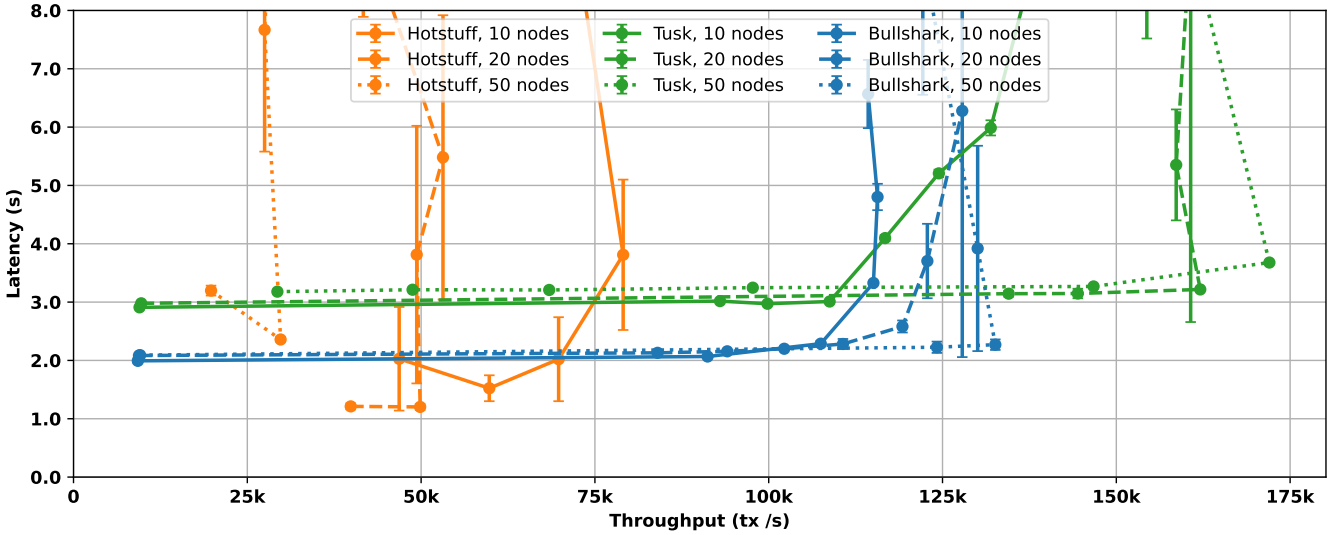

Figure 1: Comparative throughput-latency performance of HotStuff, Tusk, and BullShark. WAN measurements with 10, 20, 50 validators. No faulty validators, 500KB maximum block size and 512B transaction size.

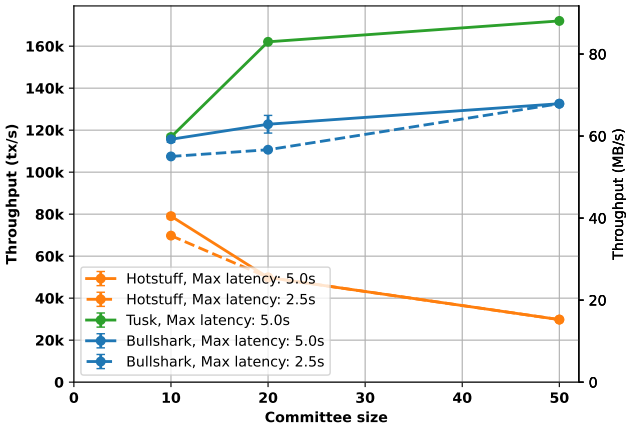

Figure 2: Maximum achievable throughput of HotStuff, Tusk, and BullShark, keeping the latency under 2.5s and 5s. WAN measurements with 10, 20, 50 validators. No faulty validators, 500KB maximum block size and 512B transaction size.

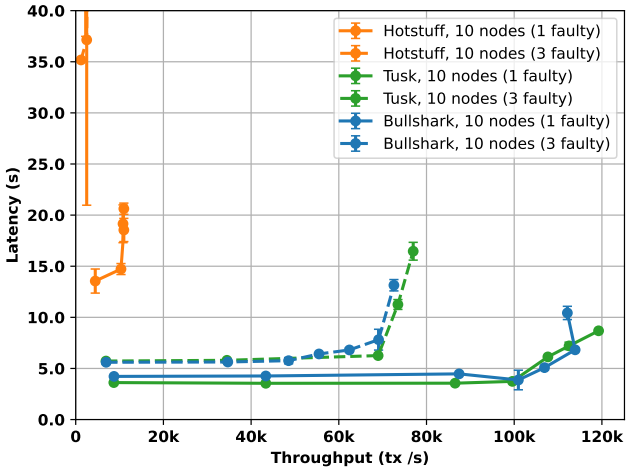

Figure 3: Comparative throughput-latency under crash-faults of HotStuff, Tusk, and BullShark. WAN measurements with 10 validators. Zero, one, and three crash-faults, 500KB maximum block size and 512B transaction size.

8 RELATED WORK

8.1 Performance comparisons

We compare BullShark with Tusk [16] and HotStuff [31]. 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 asynchronous while BullShark is partially-synchronous. 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 [11].

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 [11, 13, 14, 19, 21, 28, 30]. 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 related works are not designed to run in the

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

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.

Hotstuff-over-Narwhal [15] and Mir-BFT [29] 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 given that the extra code over Narwhal of BullShark is 200 SLOC (Alg. 5) whereas the extra code of Hotstuff is more than 4k SLOC. Additionally, BullShark can quickly adapt 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.

For Mir-BFT with transaction sizes of about 500B (similar to our benchmarks), the peak performance achieved on a WAN for 20 validators 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). 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.

Recent work [6] benchmarks crash-fault and Byzantine protocols on a LAN, yet observes orders of magnitude lower throughput than this work or Mir-BFT on WAN: 3,500 tx/sec for HotStuff and 500 tx/sec for PBFT.

DAG-based protocols The directed acyclic graphs (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 inability of 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 [27] 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 [16] 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 authorities are indistinguishable from faulty ones, and as a result the protocol proceeds without them. This creates issues around fairness and incentives, since perfectly correct, but geographically distant authorities may never be able to commit transactions submitted to them. They also require a common coin to guarantee liveness, and thus require an expensive asynchronous DKG [22]. Most asynchronous consensus protocols do not provide this DKG or entirely omit the common coin from their implementation. Further, Tusk relies on clients to re-submit a transaction if it is not sequenced in time, due to the leader being faulty. In contrast, BullShark is partially synchronous and thus does not require a expensive DKGs and common coins, and can provide timely clients confirmations similar to PBFT.

9 CONCLUSION

On the foundational level BullShark is the first DAG-based zero overhead BFT 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, asynchrnous livness, and post quantom security, while also allowing a fast-path in synchrnous periodas. Practically, the partially synchrounous version of BullShark is extrimly simple (20 LOC) and highly efficient. When implemented over the Narwhal mempool it has $2x$ the throughput of the paetially synchnrous HotStuff protocol over Narwhal and 33% lower latency than the asynchronous Tusk protocol over Narwhal.

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] Salem Alqahtani and Murat Demirbas. 2021. Bottlenecks in Blockchain Consensus Protocols. *CoRR* abs/2103.04234 (2021).
[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] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains*. Ph.D. Dissertation.
[12] 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.
[13] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.
[14] 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.
[15] George Danezis, Eleftherios Kokoris Kogias, Alberto Sonnino, and Alexander Spiegelman. 2021. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. *arXiv preprint arXiv:2105.11827* (2021).
[16] George Danezis, Eleftherios Kokoris Kogias, Alberto Sonnino, and Alexander Spiegelman. 2021. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. *arXiv preprint arXiv:2105.11827* (2021).
[17] 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.
[18] 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.
[19] 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.
[20] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. *arXiv preprint arXiv:2102.08325* (2021).
[21] 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.

[22] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. 2020. Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures.. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1751–1767.
[23] 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.
[24] Julian Loss and Tal Moran. 2018. Combining Asynchronous and Synchronous Byzantine Agreement: The Best of Both Worlds. *IACR Cryptol. ePrint Arch.* 2018 (2018), 235.
[25] Maria A Schett and George Danezis. 2021. Embedding a Deterministic BFT Protocol in a Block DAG. *arXiv preprint arXiv:2102.09594* (2021).
[26] Victor Shoup. 2000. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 207–220.
[27] 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.
[28] 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).
[29] 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
[30] 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).
[31] 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.

A PROOFS

A.1 Integrity

Theorem A.1. *For every round $r \in \mathbb{N}$, and validator $p_k \in \Pi$, a correct validator p_i outputs $a\_deliver_i(m, r, p_k)$ at most once regardless of m.*

Proof: A correct validator p_i outputs $a\_deliver(v'.block, v'.round, v'.source)$ in $order\_vertices$, where v' is a vertex from DAG_i. v' is only added to DAG_i if $r\_deliver_i(v', r, p_i)$ is called, and p_i outputs $r\_deliver_i(v', r, p_i)$ at most once (from integrity property of the reliable broadcast mechanism). Since p_i does not add any delivered vertex more than once to $deliveredVertices$, p_i will only deliver v' at most once, satisfying integrity.

A.2 Total Order

Lemma A.2. *If a correct validator p_i delivers a vertex v in wave w, then no other correct validator p_j delivers a vertex v' in wave w.*

Proof: p_i delivers v in wave w if 1) it commits v after seeing $2f + 1$ votes for v in round $2w + 1$ or 2) it commits a vertex v^* in wave $w' > w$, and then sees $f + 1$ votes for v and less than $f + 1$ votes for the fallback leader vertex in w.

We first consider case 1. By quorum intersection a correct validator p_j is guaranteed to see at least $f + 1$ votes for v in round $2w + 1$. Since there are at most $3f + 1$ total votes, and a vertex cannot be in both the steady state and the fallback modes, p_j can observe a maximum of $2f$ votes for the fallback leader vertex of w, v', which is not enough for p_j to commit v'. Let $w' > w$ be the first wave after w in which p_j commits a vertex v_j. When p_j goes back to commit earlier leaders it uses the votes from v_j to determine whether to commit v. By quorum intersection $v_j.strongEdges$ intersects the votes of p_i in wave w in at least $f + 1$ votes for v. Since there are at least $f + 1$ votes for v and at most f votes for v', p_j will deliver v.

We now consider case 2. If p_i commits v^* in wave w', then by quorum intersection p_j will see at least $f+1$ votes for v^* and at most f votes for the fallback leader of w' when it goes back to commit earlier leaders. Thus, p_j will deliver v^*. Since p_i committed v^*, p_j will also deliver v because it will also commit v^*, and therefore use the same $potentialVotes$ as p_i to commit v.

Lemma A.3. *If a correct validator p_i commits wave leader vertex v_1 in wave w_1 and after that p_i commits vertex v_2 in wave w_2, then $w_1 < w_2$.*

Proof: Leader vertices are pushed onto the stack in wave order starting from the latest wave until the last committed wave. The $order\_vertices$ procedure pops leader vertices in stack order, which means that the vertices are popped in increasing wave numbers since they were pushed by decreasing wave numbers.

Lemma A.4. *If a validator p_i commits a leader vertex v of a wave w, then for every steady state and fallback leader vertex of a wave $w' > w$, then there exists a strong path from v to them.*

Proof: If p_i commits v in wave w then p_i must have seen at least $2f + 1$ vertices with a strong path to v in round $2w + 1$. By quorum intersection, every other validator is guaranteed to have seen at least $f + 1$ of these vertices. The strong edges of any vertex in the next round will therefore contain at least $f + 1$ of these vertices.

Any future vertex in a wave $w' > w$ will have at least 1 strong path from those $f + 1$ vertices, and thus every steady state and fallback leader in wave $w' > w$ will have a strong path to v.

Theorem A.5. *If a correct validator p_i outputs $a\_deliver_i(m, r, k)$ before $a\_deliver_i(m', r', k')$ then no correct validator p_j outputs $a\_deliver_j(m', r', k')$ without first outputting $a\_deliver_j(m, r, k)$.*

Proof: By lemma A.3 each wave has only one vertex that can be committed. By lemma A.4 every correct validator commits vertices in the same order (by increasing wave number). From theorem A.5 if a correct validator p_i commits a vertex v then for every vertex u that is committed in a future wave, there is a strong path from u to v. By these properties any two correct validators will commit the same vertices in the same order. Once a validator commits a vertex then it also commits its causal history in some deterministic order, which is the same for all validators, meaning that correct validators will deliver vertices in the same order.

A.3 Agreement

Lemma A.6. *The probability that for a wave w the commit rule is met is upper bounded by $\frac{2f+1}{3f+1} + \epsilon$*

Proof: The fallback wave realizes the common-core abstraction for which there is a subset V of at least $2f+1$ values such that for every correct validator there is a common core of at least $2f + 1$ input values that appear in the returned sets of all the correct validators that complete the common-core abstraction. The fallback wave has three rounds of each validator reliably broadcasting its input value, and then waiting for $2f + 1$ input values. These three rounds can be mapped to the three stages of the common-core algorithm as shown in DAG-Rider. Now since we have a set of $2f + 1$ common vertices, the probability of a commit rule being satisfied is $\frac{2f+1}{3f+1} + \epsilon$.

Lemma A.7. *For every correct validator p_i and for every wave w, the expected number of waves from w until the fallback commit rule is met is bounded by $\frac{3}{2} + \epsilon$.*

Proof: For each wave w, the probability that for a correct validator p_i the commit rule is met is at least $\frac{2f+1}{3f+1} + \epsilon$ from lemma A.6. Thus the expected number of waves is bounded by $\frac{3}{2} + \epsilon$.

Theorem A.8. *If a correct validator p_i outputs $a\_deliver_i(b, r, p_k)$ then every other correct validator eventually outputs $a\_deliver_j(b, r, p_k)$.*

Proof: If p_i delivers a vertex u then it must be in the causal history of some vertex v, which the steady state or fallback leader of some wave. By lemma A.7, every other correct validator p_j that has not committed v yet will eventually with probability 1, have a wave $w' > w$, in which the commit rule is met. By the total order property once v is committed, v's causal history will also be committed in the same order, including u.

A.4 Validity

Theorem A.9. *If a correct validator p_k calls $a\_bcast_k(m, r)$, then every correct validator p_i eventually outputs $a\_deliver_i(m, r, k)$.*

Proof: When p_k calls $a\_bcast_k(m, r)$, it reliably broadcasts a vertex v in round r, and makes sure it has a path to all the vertices in rounds $r' < r$. If this is not satisfied by strong edges in $r - 1$, then it adds weak edges to vertices in rounds $r' < r - 1$ to satisfy this. Therefore v will eventually be included in the causal history of all correct validators, and therefore in the causal history of a committed leader vertex. Thus, v will be delivered.