BullShark: DAG BFT Protocols Made Practical

Alexander Spiegelman
sasha.spiegelman@gmail.com
Aptos

Alberto Sonnino
alberto@sonnino.com
Mysten Labs

Neil Giridharan
giridhn@berkeley.edu
University of California, Berkeley

Lefteris Kokoris-Kogias
Lefteris2k@gmail.com
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 and view-synchronization 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 transactions 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.

CCS CONCEPTS
• Security and privacy → Distributed systems security.

KEYWORDS
Consensus protocol, Byzantine Fault Tolerant

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:
Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. BullShark: DAG BFT Protocols Made Practical. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS ’22). ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3559361

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 [9, 19, 21, 25, 33, 42] proposed an elegant way to separate between the dissemination 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 allows 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 [9] 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 [12] to avoid equivocation and an honest party advances to the next round once it reliably delivers \( 2f + 1 \) vertices in the current round. 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 [15] 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 practice. 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 [13, 43] 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].

To the best of our knowledge, this paper is the first to optimize the DAG-Based BFT approach to the partially synchronous communication setting. First, we propose BullShark, which preserves all the theoretical 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 roundtrips 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). In addition, BullShark is built on top of Narwhal and thus inherit all its practical benefits (e.g., decoupling data dissemination from the DAG construction and efficient reliable broadcast implementation).

Second, based on BullShark’s fast path, we present an eventually synchronous variant of BullShark, which is the first partially 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 or view synchronization mechanism after a faulty leader. 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 an existing practical limitation of DAG-based protocols of having to choose between fairness and garbage collection. BullShark garbage collects vertices belonging to old DAG rounds, and at the same time provide fairness during synchronous periods. As an evidence to its practicality, the partially synchronous version of BullShark has already been productionized by Mysten labs and is currently being integrated by Aptos.

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 in expectation (matching DAG-Rider) in asynchrony.

- We simplify BullShark to perform only in partial synchrony. This version of BullShark results in a significantly simpler partially 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 TECHNICAL CHALLENGES.

In order to design and implement BullShark we had to solve a number of theoretical and practical challenges.

Theoretical challenges. 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). This works perfectly for asynchronous consensus, but unfortunately cannot guarantee deterministic liveness during synchronous periods [20], 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 leaders. Note that this is inherent to any deterministic protocol. We considered and evaluated two alternatives (see Appendix B) and decided to embed timeouts into the DAG construction as it provided better performance. In a nutshell, if the first $2f + 1$ messages in a round do not contain one from the leader, then parties wait for a timeout or a message from the leader before advancing to the next round.

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 the 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 broadcasted 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 histories of vertices in their DAGs, 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 proofs) 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.

**Practical challenges.** 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. We started from Narwhal’s open source code base and adopt their approach to decouple data from metadata to implement an 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 implementations in asynchronous networks 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.

## 3 PRELIMINARIES

### 3.1 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 protocol description we assume reliable links between honest parties. That is, all messages among honest parties eventually arrive $^1$. Moreover, we assume that recipients can verify the senders identities. We assume a known $\Delta$ 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 $\Delta$ 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 $^2$

---

$^1$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.

### 3.2 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 $r_{\text{bcast}}(m, r)$, where $m$ is a message and $r \in \mathbb{N}$ is a round number. Every party $p_i$ has an output $r_{\text{deliver}}(m, r, p_k)$, where $m$ is a message, $r$ is a round number, and $p_k$ is the party that called the corresponding $r_{\text{bcast}}(m, r)$. The reliable broadcast abstraction guarantees the following properties:

- **Agreement** If an honest party $p_i$ outputs $r_{\text{deliver}}(m, r, p_k)$, then every other honest party $p_j$ eventually outputs $r_{\text{deliver}}(m, r, p_k)$.
- **Integrity** For each round $r \in \mathbb{N}$ and party $p_k \in \Pi$, an honest party $p_i$ outputs $r_{\text{deliver}}(m, r, p_k)$ at most once regardless of $m$.
- **Validity** If an honest party $p_k$ calls $r_{\text{bcast}}(m, r)$, then every honest party $p_i$ eventually outputs $r_{\text{deliver}}(m, r, p_k)$.

**Global perfect coin** An instance $w, w \in \mathbb{N}$, of the coin is invoked by party $p_i \in \Pi$ by calling $\text{choose\_leader}(w)$. 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 $\text{choose\_leader}(w)$. The global perfect coin has the following guarantees:

- **Agreement** If two honest parties $p_i, p_j$ call $\text{choose\_leader}(w)$ and $\text{choose\_leader}(w)$ with respective return values $p_1$ and $p_2$, then $p_1 = p_2$.
- **Termination** If at least $f + 1$ honest parties call $\text{choose\_leader}(w)$, then every $\text{choose\_leader}(w)$ call eventually returns.
- **Unpredictability** As long as less than $f + 1$ honest parties call $\text{choose\_leader}(w)$, the return value is indistinguishable from a random value except with negligible probability $\epsilon$. Namely, the probability $p_r$ that the adversary can guess the returned party $p_j$ of the call $\text{choose\_leader}(w)$ is $p_r \leq \Pr[X_w = p_j] + \epsilon$.
- **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 [11, 29, 34] can be found in [14, 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.

### 3.3 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 $a_{\text{bcast}}(m, r)$ and $a_{\text{deliver}}(m, r, p_k)$, 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 3.1 (Byzantine Atomic Broadcast).** Each honest party \( p_i \in \Pi \) can call \( \text{a\_bcast}(m, r) \) and output \( \text{a\_deliver}_i(m, r, p_k) \), \( p_k \in \Pi \). A Byzantine Atomic Broadcast protocol satisfies reliable broadcast (agreement, integrity, and validity) as well as:

- **Total order** If an honest party \( p_i \) outputs \( \text{a\_deliver}_i(m, r, p_k) \) before \( \text{a\_deliver}_i(m', r', p'_k) \), then no honest party \( p_j \) outputs \( \text{a\_deliver}_j(m', r', p'_k) \) before \( \text{a\_deliver}_j(m, r, p_k) \).

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 7, 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 [8], it supports a separation between the total order mechanism and transaction execution. Transaction validation can therefore be done as part of the execution [8] before applying to the SMR.

### 4 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 deterministically achieves better latency in synchronous periods. Therefore, introducing timeouts into the system is unavoidable [20]. We considered and evaluated two alternatives (see Appendix B for more details) and decided to integrate 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 4.1. We describe our DAG construction in Section 4.2.

#### 4.1 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 views 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 basic utilities are specified in Algorithm 1. For each party \( p_i \), we denote \( p_i \)’s local view of the DAG as \( \text{DAG}_i \), which is represented by an array of sets of vertices \( \text{DAG}_i(r) \). Vertices are created via the \( \text{create\_new\_vertex}(r) \) 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 \( \text{a\_bcast} \) 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 vertices associated with round \( r \) and the set **weak edges** contains up to \( f \) references to vertices in rounds \( < r \) 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 \( \text{DAG}_i(r) \) 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 \( |\text{DAG}_i(r)| \leq n \).

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

The procedure \( \text{get\_fallback\_vertex\_leader} \) 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 \( \text{get\_first\_steady\_vertex\_leader} \) and \( \text{get\_second\_steady\_vertex\_leader} \) 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.

#### 4.2 Our DAG protocol

A detailed pseudocode is given in Algorithm 2. Each party \( p_i \) maintains three local variables: \( \text{round} \) stores the last round in which \( p_i \) broadcast a vertex, \( \text{buffer} \) stores vertices that where reliably delivered but not yet added to the DAG, and \( \text{wait} \) 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 \( \text{try\_add\_to\_DAG} \). 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
Algorithm 1 Data structures and basic utilities for party $p_i$

Local variables:
- struct vertex $v$:
  - $v$.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
  - $DAG(\cdot)$ - An array of sets of vertices, initially: $DAG[\cdot] \rightarrow$ predefined hardcoded set of $2f + 1$ "genesis" vertices
- ∀ $j \geq 1$, $DAG[j] \rightarrow \{}$
- blocksToPropose - A queue, initially empty, $p_i$ enqueues valid blocks of transactions from clients

1: procedure path($v$, $u$)
2: return exists a sequence of $k \in \mathbb{N}$, vertices $v_1, v_2, \ldots, v_k$ s.t.
3: $v_1 = v$, $v_k = u$, and ∀ $i \in [2, k]$: $v_i \in \bigcup_{j=1}^{\infty} DAG[r] \land (v_j \in v_{i-1}.weakEdges \cup v_{i-1}.strongEdges)$

3: procedure strong_path($v$, $u$)
4: return exists a sequence of $k \in \mathbb{N}$, vertices $v_1, v_2, \ldots, v_k$ s.t.
5: $v_1 = v$, $v_k = u$, and ∀ $i \in [2, k]$: $v_i \in \bigcup_{j=1}^{\infty} DAG[r] \land v_i \in v_{i-1}.strongEdges$

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

13: procedure set_weak_edge($v$, round) $\triangleright$ Add edges to orphan vertices
14: $v$.weakEdges $\leftarrow \{\}$
15: for $r = \text{round} - 2$ down to 1 do
16:   for every $u \in DAG[\cdot] | s.t. \rightarrow path(v, u)$ do
17:     $v$.weakEdges $\leftarrow v$.weakEdges $\cup \{u\}$
18: procedure get_fallback_vertex_leader($w$)
19: $p \leftarrow \text{choose_leader}(w)$
20: return get_vertex($p, 4w - 3$)
21: procedure get_first_steady_vertex_leader($w$)
22: $p \leftarrow \text{get_first_predefined_leader}(w)$
23: return get_vertex($p, 4w - 3$)
24: procedure get_second_steady_vertex_leader($w$)
25: $p \leftarrow \text{get_second_predefined_leader}(w)$
26: return get_vertex($p, 4w - 1$)
27: procedure get_vertex($p, v$)
28: if $\exists w \in DAG[\cdot]$ s.t. $w$.source = $p$ then
29: return $v$
30: return $\bot$

$v$ is added to the DAG, the high-level BAB protocol is invoked, via the try_ordering($\cdot$) 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 collecting enough votes to commit an honest leader after GST. Therefore, before trying to advance (via try_advence_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-state leader, respectively. In the full paper [37], we prove that after GST timeouts never expire for honest leaders and the DAG advances in network speed.

5 THE BULLSHARK PROTOCOL

In this section we present a detailed description of the full BullShark protocol. A stand alone description of the deterministic eventually synchronous BullShark protocol is given in Section 6 and more details can be found in [38]. 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
Unlike DAG-Rider, which has one potential leader in every wave, its local view of the DAG, similarly to DAG-Rider, to interpret the DAG, each party totally orders leaders’ causal histories. In Section 6 we preset parties need to make sure parties cannot vote for both types at the same round. Illustration of BullShark can be found in Figure 1. We divide the protocol description into two parts. In Section 5.1 we describe the commit rule of each leader, and in Section 5.2 we explain how parties totally order leaders’ causal histories. In Section 6 we present an eventually synchronous version of BullShark and in Section 7 we discuss the details of our garbage collection mechanism. For space limitations, we provide formal proofs for both versions on BullShark in the full paper [37].

### 5.1 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. 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.

---

**Algorithm 2** DAG construction, protocol for process $p_i$

```plaintext
Local variables:
round ← 1; buffer ← {}; wait ← true

31: upon $r_{delivers}(o, r, p)$ do
32: if $o.source = p \land o.round = r \land |o.strongEdges| \geq 2f + 1$ then
33: if $\neg try\_add\_to\_dag(o)$ then
34: buffer ← buffer $\cup$ [o]
else
35: for $v \in$ buffer : o.round $\leq$ r do
36: try_add_to_dag(o)
37: if r = round then
38: w ← $\{r/4\}$ — steady state wave number
39: if $r \mod 4 = 1 \land (\neg wait \lor \exists o \in DAG[r] : o.source = get\_first\_steady\_vertex\_leader(w))$ then
40: try Advance_round()
41: if $r \mod 4 = 3 \land (\neg wait \lor \exists o \in DAG[r] : o.source = get\_second\_steady\_vertex\_leader(w))$ then
42: try Advance_round()
43: if $r \mod 4 = 0 \land (\neg wait \lor \exists U \subseteq DAG[r] : |U| = 2f + 1$ and $\forall u \in U, u.source \in steady\_voters(w)) \land strong\_path(u, get\_second\_steady\_leader(w))$ then
44: try Advance_round()
45: if $r \mod 4 = 2 \land (\neg wait \lor \exists U \subseteq DAG[r] : |U| = 2f + 1$ and $\forall u \in U, u.source \in steady\_voters(w)) \land strong\_path(u, get\_first\_steady\_leader(w))$ then
46: try Advance_round()
47: upon timeout do
48: wait ← false
49: try Advance_round()

51: procedure try_add_to_dag(o)
52: if $\forall v' \in o.strongEdges \cup o.weakEdges : v' \notin \bigcup\_k DAG[k]$ then
53: $DAG[v.round] ← DAG[v.round] \cup$ [o]
54: if $|DAG[v.round]| \geq 2f + 1 \land o.round >$ round then
55: round ← round + 1; start timer; wait ← true — Synchronize waves
56: broadcast_vertex(o.round)
57: buffer ← buffer $\cup$ [o]
58: try ordering(o)
59: return true
60: return false

61: procedure try Advance_round()
62: if $|DAG[round]| \geq 2f + 1$ then
63: round ← round + 1; start timer; wait ← true
64: broadcast_vertex(round)

65: procedure broadcast_vertex(r)
66: $v ← create\_new\_vertex(r)$
67: try add to dag(o)
68: r. leader(o, r)
```

---

**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 have observed at least f + 1 votes.
Corollary 3.1 BullShark part 1: $p_i$'s alg. to update parties vote type

**Local variables:**
- steadyVoters[w] \[\triangleright=\] Steady, fallbackVoters[w] \[\triangleright=\] {}  
- For every $j > 1$, steadyVoters[f] \[\triangleright=\] {}  

69: upon a_bcast, $(b, r)$ do  
70: blocksToPropose.enqueue(b)  

71: procedure try_ordering(o)  
72: w \[\leftarrow\] (o.round/d)  
73: votes \[\leftarrow\] o.strongEdges  
74: if o.round \mod 4 = 1 then  
75: determine_party_vote_type(v.source, votes, w)  
76: else if o.round \mod 4 = 3 then  
77: try_steady_commit(votes, get_first_steady_vertex_leader(w), w)  

78: procedure determine_party_vote_type(p, votes)  
79: \[\forall\left(v \in \text{steadyVoters}[w]ight)\text{, steadyVoters}[w] \cup \{p}\]  
80: if there exists $v$ s.t. $v \in \text{steadyVoters}[w]$ and $\forall v' \in \text{steadyVoters}[w]$  
81: \[\exists v' \in \text{steadyVoters}[w] \text{ and }\]  
82: try_steady_commit(votes, v, w)  
83: else  
84: fallbackVoters[w] \[\leftarrow\] fallbackVoters[w] \cup \{p\}  

85: procedure try_steady_commit(votes)  
86: if $\forall v' \in \text{votes}$ s.t. $v' \in \text{steadyVoters}[w] \land \left|\text{strong_paths}(v', v)\right| \geq 2f + 1$ then  
87: commit_leader(v)  
88: return true  
89: return false  

90: procedure try_fallback_commit(votes)  
91: if $\forall v' \in \text{votes}$ s.t. $v' \in \text{fallbackVoters}[w] \land \left|\text{strong_paths}(v', v)\right| \geq 2f + 1$ then  
92: commit_leader(v)  
93: return true  
94: return false

particular, parties can see what information other parties had when they interpreted the DAG, and decide accordingly.

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 reliable broadcast properties, all parties see the same causal history of vertex v, and thus agree on $p_j$'s voting type in round w (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 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 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.

5.2 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.

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 $v$ in the round has at least $f + 1$ votes while the other has at most $f$. If this is
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. The presentation here is based on the terminology of Section 5. A stand alone description can be found in [38]. An intuitive illustration can be found in Appendix A and in [3].

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). The full paper [37] we give a formal proof of Safety and Liveness. 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 then all honest parties will wait for it before advancing to the fourth round, and thus all honest will see at least $2w + 1$ votes for the second leader in w and commit it.

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

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

95: procedure commit_leader(o)
96:    leaderStack.push(o)
97:    r ← round + 2  \text{ » There is a potential leader to commit every two rounds}
98:    while $r > $committedRound do
99:        w ← \text{[r/4]}
100:        ssPotentialVotes ← \{ $v \in \text{DAG}[r + 1] : \text{strong}_\text{path}(v, v') \}
101:        if $r \mod 4 = 2$ then
102:            \text{» two potential leaders in this round}
103:            $\omega_i \leftarrow \text{get_first_steady_vertex_leader}(v) \text{ w}
104:            $\omega_j \leftarrow \text{fallback_voter_leader}(w)
105:            ssVotes ← \{ $v \in \text{ssPotentialVotes} : \omega_i \text{.source} \in\text{steadyVotes}[w] \land \text{strong}_\text{path}(v', \omega_i) \}
106:            if $r \mod 4 = 3$ then
107:                \text{» fallback leader could not be committed since there at least 2f + 1 steady-state votes types in this wave}
108:            else
109:                fbVotes ← \{ $v \in \text{fbPotentialVotes} : \omega_j \text{.source} \in\text{fallowVotes}[w] \land \text{strong}_\text{path}(v', \omega_j) \}
110:            end
111:        \text{else}
112:            $\omega_i \leftarrow \text{get_second_steady_vertex_leader}(w)
113:            ssVotes ← \{ $v \in \text{ssPotentialVotes} : \omega_i \text{.source} \in\text{steadyVotes}[w] \land \text{strong}_\text{path}(v', \omega_i) \}
114:            $\omega_j \leftarrow \omega_i
115:            if $\text{ssVotes} \geq f + 1$ then
116:                leaderStack.push($\omega_i$)
117:            \text{else}
118:                if $\text{ssVotes} < f + 1$ then
119:                    leaderStack.push($\omega_j$)
120:        end
121:        r ← $r - 2$
122:        committedRound ← $r\text{.round}$
123:        order_vertices()
124:    end
125:    \text{procedure order_vertices()}
126:    \text{while }leadersStack.isEmpty() \text{ do}
127:        $v \leftarrow \text{leaderStack.pop()}$
128:        verticesToDeliver ← \{ $v' \in \bigcup_{r=0}^\infty \text{DAG}[r] : \text{path}(v, v') \land \omega_i \notin \text{deliveredVertices} \}
129:        \text{for every } v' \in \text{verticesToDeliver in some deterministic order do}
130:            \text{output } a \leftarrow \text{deliver}(v'.block, v'.round, v'.source)
131:            deliveredVertices ← deliveredVertices $\cup \{ v' \}$
132:        end

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 — is guaranteed that none of them have been committed.

As we prove in the full paper [37], 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_vertices(). 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.

6 EVENTUALLY SYNCHRONOUS BULLSHARK

In this section we present an eventually synchronous version of the Bullshark protocol. This protocol is embarrassingly simple, and as

7 GARBAGE COLLECTION IN BULLSHARK

One of the main practical challenges in DAG-based BFT protocols 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.

For example, HashGraph [9] 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, makes use of weak links to refer to unordered blocks in previous rounds, which guarantees every block is eventually ordered. The solution works well in theory, but garbage collection is an issue.

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 \( \text{o} \) 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 7.1.** If an honest party \( p_k \) calls \( r_{bcast}(m, r) \) after GST, then every honest party \( p_i \) eventually outputs \( r_{deliver}(m, r, k) \).

For the garbage collection mechanism we add a timestamp for every vertex. That is, an honest party specifies in \( v.ts \) the time when it broadcast its vertex \( v \). In addition, parties maintain a garbage collection round, \( G\text{Ground} \), 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 \( \Delta \). 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 \( v \)'s 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 \( 3\Delta \) the round is garbage collected.

Since by the properties of the underlying 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. Below we argue that when announced with the above garbage collection, BullShark satisfies Definition 7.1 while preserving bounded memory.

**Algorithm 6 Garbage collection. Algorithm for party \( p_i \).**

| Local variables: | G\text{Round} ← 0 |
|------------------|------------------|
| 1. procedure order_vertices() | 2. while ~leadersStack.isEmpty() do |
| 3. \( v ← \text{leadersStack.pop()} \) | 4. if \( \text{n.round} > 1 \) then |
| 5. \( \text{parents} ← \{ u ∈ G\text{AG}[\text{n.round} - 1] | \text{path}(v, u) \} \) | 6. leaderTS ← median(\{v.ts | v ∈ parents\}) |
| 7. \( \text{verticesToDoDeliver} ← \text{parents} \cup \{ v \} \) | 8. else |
| 9. \( \text{verticesToDoDeliver} ← \{ v \} \) | 10. \( r ← \text{G\text{Round}} + 1 \) |
| 11. while \( r < \text{n.round} - 1 \) do | 12. \( \text{candidates} ← \{ u ∈ G\text{AG}[r] | \text{path}(v, u) \} \) |
| 13. \( \text{candidatesTS} ← \text{median}(\{v.ts | v ∈ \text{candidates} \}) \) | 14. \( \text{verticesToDoDeliver} ← \text{verticesToDoDeliver} \cup \text{candidates} \setminus \text{deliveredVertices} \) |
| 15. if leaderTS - \text{candidatesTS} > 3\Delta then | 16. \( G\text{Round} ← r \) |
| 17. \( \text{G\text{AG}[r]} ← \{ \} \) | 18. \( r ← r + 1 \) |
| 19. for every \( v' ∈ \text{verticesToDoDeliver} \) do | 20. \( \text{output} \_\text{a_deliver}(v', \text{block}, r', \text{round}, \text{source}) \) |
| 21. \( \text{deliveredVertices} ← \text{deliveredVertices} \cup \{ v' \} \) |

**Bounded memory.** In the full paper [37] 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 \( p_1 \) be a party that broadcasts 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 \( p_1 \) be the first party that advances to round \( r \). In the full paper [37] 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, \( p_1 \) 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 \( v'.\text{strongEdges} \) 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 \( v' \).

**8 IMPLEMENTATION**

We implement a networked multi-core eventually synchronous 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 4.2. Additionally, it provides well-documented benchmarking scripts to measure performance in various conditions, and it is close to

\(^3\)https://github.com/facebookresearch/narwhal
a production system (it provides real networking, cryptography, and persistent storage). It is implemented in Rust, uses tokio for asynchronous networking, ed25519-dalek for elliptic curve based signatures, and data-structures are persisted using Rocksdb. 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\(^4\) along with any Amazon web services orchestration scripts and measurements data to enable reproducible results\(^5\).

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 [10].

We deploy a testbed on AWS, using m5.\(8x\)large\(\text{\_}x\)large 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 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 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.

9.2 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 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 [39] 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. 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

\(^4\)https://github.com/asonnino/narwhal/tree/bullshark
\(^5\)https://github.com/asonnino/narwhal/tree/bullshark/benchmark/data

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. BullShark’s selling point over Tusk is 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.
We limit our comparison to these two systems, thus omitting a number of important related works such as [13, 16, 17, 24, 26, 40, 42]. A practical comparison with those systems would hardly be fair as (e.g., have no synchronizer), or are internally sized to process empty implementations of prior works are not designed to run in the WAN and transactions and are thus not adapted to the 512B transaction size.

10 RELATED WORK

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

**Performance comparisons:** We compare BullShark with Tusk [19] and HotStuff [43]. 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 a partially-synchronous variant of the Tendermint [13] protocol, which runs at the heart of a number of projects [1, 2, 4–6].

We aim to compare BullShark with related systems as fairly as possible. An important reason for selecting Tusk and HotStuff 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 as 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 [13, 16, 17, 24, 26, 40, 42]. 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 [41] 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 (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 [9] 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 Kurasawae 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 [7] protocol for fallback. As a result it reduces the latency cost of BDT significantly, but cannot be generalized to a plug-and-play framework.

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.
A PARTIALLY SYNCHRONOUS BULLSHARK ILLUSTRATION

Figure 5 illustrates the partially synchronous Bullshark protocol for $n = 4$ and $f = 1$. Each odd round in the DAG has a predefined leader vertex (highlighted in solid green) and the goal is to first decide which leaders to commit. Then, to totally order all the vertices in the DAG, a party goes one by one over all the committed leaders and deterministically orders their causal histories.

Each vertex in an even round can contribute one vote for the previous round leader. In particular, a vertex in round $r$ votes for the leader of round $r - 1$ if there is an edge between them. The commit rule is simple: a leader is committed if it has at least $f + 1$ votes. In Figure 5, L3 is committed with 3 votes, whereas L1 and L2 have less then $2 = f + 1$ votes and are not committed.

Due to the asynchronous nature of the network, the local views of the DAG might differ for different parties. That is, some vertices might be delivered and added to the local view of the DAG of some of the parties but not yet delivered by the others. Therefore, even though some validators have not committed L1, others might have.

To guarantee all parties commit the same leaders, Bullshark relies on quorum intersection:

Since the commit rule requires $f + 1$ votes and each vertex in the DAG has at least $n - f$ edges to vertices from the previous round, it is guaranteed that if some validator commits a leader $L$ then all future leaders will have a path to at least one vertex that voted for $L$, and thus will have a path to $L$.

Therefore: If there is no path to a leader $L$ from a future leader, then no party committed $L$ and it is safe to skip $L$.

The logic to order leaders is the following: when a leader $i$ is committed, the party checks if there is a path between leader $i$ to leader $i - 1$. If this is the case, leader $i - 1$ is ordered before leader $i$ and the logic is recursively restarted from $i - 1$.

Otherwise, leader $i - 1$ is skipped and the party checks if there is a path between $i$ to $i - 2$. If there is a path, leader $i - 2$ is ordered before $i$ and the logic is recursively restarted from $i - 2$. Otherwise, leader $i - 2$ is skipped and the process continues in the same way. The process stops when it reaches a leader that was previously ordered. In Figure 5, leaders L1 and L2 do not have enough votes to be committed and once the party commits L3 it has to decide whether to order L1 and L2. Since there is no path from L3 to L2, L2 can be skipped. However, since there is a path between L3 and L1, L1 is ordered before L3. Now, to totally order the vertices of the DAG, the party first orders the causal history of L1 (nothing to order in this example) by some deterministic rule and then orders the causal history of L3.

B LOGICAL VS PHYSICAL DAG

As mentioned above, to provide deterministic fast path, introducing timeouts is unavoidable [20]. After implementing and evaluating two alternatives, we decided to embed the timeouts into the DAG construction as described above. Intuitively, it might look inefficient as the DAG does not advance in network speed, but as we shortly explain, it is the other way round.

The other approach we consider is a virtual consensus DAG layer on top of the physical DAG. In this case the physical level has no timeouts and is very similar to the DAG construction in DAG-Rider, which advances rounds in networks speed once $2f + 1$ nodes in the current round are delivered. To encode timeouts, some of the nodes in the physical DAG have “consensus” headers indicating that they belong to the virtual level. The logic to advance consensus rounds is almost similar to the one described in Alg 2. That is, consensus nodes indicate in their consensus header to which virtual nodes they refer as parents. This virtual nodes can be in arbitrary physical DAG rounds but they are at exactly one less ($r - 1$) consensus round. As a result, now timeouts are only needed at the virtual level and do not interfere with the physical DAG advancement. The only difference from Alg 2 is that weak links are not required on the virtual level since the weak links on the physical level already guarantee the validity property. All in all, the physical DAG advances in network speed and the virtual DAG provides the functionality required by the Bullshark consensus protocol.

We implemented and evaluated this logical DAG construction, however, the results were not encouraging (around 50% latency increase without any significant throughput benefit). After investigation we attributed this to two main reasons:

- Since BullShark is built on top of Narwhal, it inherent the data dissemination decoupling from the DAG construction. That is, data is disseminated at network speed regardless of the DAG construction, which contains only metadata. Therefore, if the DAG advances rounds slower, then each vertex in the DAG simply contains more metadata and the throughput is not compromised.
- The logical split between virtual and physical DAG introduces a decoupling between delays/timeouts for the consensus messages and delays for the block creation. This results to a common pattern where a physical DAG blocks is created milliseconds before a vote is ready to be cast, but the vote missed the block and needs to wait for the next round to be cast. This introduces a small delay per vote but since we need $2f+1$ votes to commit a consensus round the latency of the DAG moves from the median latency to the tail-latency of the 66th percentile.
- Moreover, the smaller the DAG the less resources are required to manage it. For example, less memory to store it and less bandwidth to construct it.
REFERENCES

[1] 2022. Ceo. https://ceo.org. (2022).
[2] 2022. Cypherium. https://www.cypherium.io. (2022).
[3] 2022. DAG meets BFT. https://decentralizedthoughts.github.io/2022-06-28-DAG-meets-BFT. (2022).
[4] 2022. Diem. https://www.diem.com. (2022).
[5] 2022. Flow. https://www.onflow.org. (2022).
[6] 2022. Thunder. https://www.thundercore.com/. (2022).
[7] 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 (PODC). 337–346.
[8] Eli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Eneyeart, Christopher Ferris, Gennady Lavestman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Proceedings of the thirteenth EuroSys conference. 1–15.
[9] Leemon Baird. 2016. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. Swirls Tech Reports SWIRLDS-TR-2016-01, Tech. Rep. (2016).
[10] Shehar Bano, Alberto Sonnino, Andrey Chzursin, Dmitri Perelman, and Dahlia Malkhi. 2021. Twins: BFT Systems Made Robust. In Principles of Distributed Systems.
[11] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In International conference on the theory and application of cryptography and information security. Springer, 514–532.
[12] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. Information and Computation 75, 2 (1987), 130–143.
[13] Ethan Buchman. 2016. Tendermint: Byzantine fault tolerance in the age of blockchains. Ph.D. Dissertation.
[14] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random oracles in Con- stant-time: Practical asynchronous Byzantine agreement using cryptography. Journal of Cryptology 18, 3 (2005), 219–246.
[15] Christian Cachin and Stefano Tersario. 2005. Asynchronous verifiable information dispersal. In 24th IEEE Symposium on Reliable Distributed Systems (SRDS’05). IEEE, 191–201.
[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 Zhoushan Xiang. 2021. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In PC ’21: Proceedings of the 26th International Conference on Financial Cryptography.
[23] Rachid Guerraoui, Nikola Knezěv, Vivien Quêma, and Marko Vukolic. 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, Lefteris Kokoris-Kogias, Oded Naveh, and Alexander Spiegelman. 2021. All you need is dag. In Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing. 165–175.
[26] Lefteris Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Kohfi, 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 Conference 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), 815–829.
[29] Benoît Libert and Marc Joye. 2021. On the forward secure disk encryption. In International Conference on Financial Cryptography and Data Security. Springer, 507–527.
[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, Zhienliang 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). 605–620.
[37] Alexander Spiegelman, Neil Gidirharian, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG bft protocols made practical (full paper). arXiv preprint arXiv:2201.06577 (2022).
[38] Alexander Spiegelman, Neil Gidirharian, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: The Partially Synchronous Version. arXiv preprint arXiv:2009.06933 (2022).
[39] 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 f"ur Informatik.
[40] Chrysovala Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolic. 2019. Mir-BFT: High-Throughput Robust BFT for Decentralized Networks. arXiv preprint arXiv:1906.05552 (2019).
[41] Chrysovala 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
[42] 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).
[43] Maosan 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.