# All You Need is DAG

Idit Keidar
Technion

Eleftherios Kokoris-Kogias
IST Austria and Novi Research

Oded Naor*
Technion

Alexander Spiegelman
Novi Research

## ABSTRACT

We present *DAG-Rider*, the first asynchronous Byzantine Atomic Broadcast protocol that achieves optimal resilience, optimal amortized communication complexity, and optimal time complexity. DAG-Rider is post-quantum safe and ensures that all values proposed by correct processes eventually get delivered. We construct DAG-Rider in two layers: In the first layer, processes reliably broadcast their proposals and build a structured Directed Acyclic Graph (DAG) of the communication among them. In the second layer, processes locally observe their DAGs and totally order all proposals with no extra communication.

## CCS CONCEPTS

• **Theory of computation → Distributed algorithms**; • **Security and privacy → Distributed systems security**.

## KEYWORDS

Asynchrony; Atomic Broadcast; Byzantine SMR; Quantum safe

**ACM Reference Format:**
Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing (PODC '21), July 26–30, 2021, Virtual Event, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3465084.3467905

## 1 INTRODUCTION

The amplified need in scalable geo-replicated Byzantine fault-tolerant reliability systems has motivated an enormous amount of study on the Byzantine State Machine Replication (SMR) problem [17, 31]. Many variants of the problem were defined in recent years [28, 32, 43] to capture the needs of blockchain systems. To address the fairness issues that naturally arise in interorganizational deployments, we focus on the classic long-lived Byzantine Atomic Broadcast (BAB) problem [12, 19], which in addition to total order and progress also guarantees that *all* proposals by correct processes are eventually included.

Up until recently, asynchronous protocols for the Byzantine consensus problem [12, 16, 26] have been considered too costly or complicated to be used in practical SMR solutions. However, two recent single-shot Byzantine consensus papers, VABA [1] and later Dumbo [35], presented asynchronous solutions with (1) optimal resilience, (2) expected constant time complexity, and (3) optimal quadratic communication and optimal amortized linear communication complexity (for the latter). In this paper, we follow this recent line of work and present *DAG-Rider*: the first asynchronous BAB protocol with optimal resilience, optimal round complexity, and optimal amortized communication complexity. In addition, given a perfect shared coin abstraction, our protocol does not use signatures and does not rely on asymmetric cryptographic assumptions. Therefore, when using a deterministic threshold-based coin implementation with an information theoretical agreement guarantee [13, 34], the safety properties of our BAB protocol are post-quantum secure.

*Overview.* We construct *DAG-Rider* in two layers: a communication layer and a zero-overhead ordering layer. In the communication layer, processes reliably broadcast their proposals with some metadata that help them form a *Directed Acyclic Graph (DAG)* of the messages they deliver. That is, the DAG consists of rounds s.t. every process broadcasts at most one message in every round and every message has $O(n)$ references to messages in previous rounds, where $n$ is the total number of processes. The ordering layer does not require any extra communication. Instead, processes observe their local DAGs and with the help of a little randomization (one coin flip per $O(n)$ decisions on values proposed by different processes) locally order all the delivered messages in their local DAGs.

A nice feature of DAG-Rider is that the propose operation is simply a single reliable broadcast. The agreement property of the reliable broadcast ensures that all correct processes eventually see the same DAG. Moreover, the validity property of the reliable broadcast guarantees that all broadcast messages by correct processes are eventually included in the DAG. As a result, in contrast to the VABA and Dumbo protocols that retroactively ignore half the protocol messages and commit one value out of $O(n)$ proposals, DAG-Rider does not waste any of the messages and all proposed values by correct processes are eventually ordered (i.e., there is no need to re-propose).

*Complexity.* We measure time complexity as the asynchronous time [16] required to commit $O(n)$ values proposed by different correct processes, and we measure communication complexity by the number of bits processes send to commit a single value. To compare DAG-Rider to the state-of-the-art asynchronous Byzantine agreement protocols, we consider SMR implementations that run an unbounded sequence of the VABA or Dumbo protocols to independently agree on every slot. To compare apples to apples in respect to our time complexity definition, we allow VABA and

*Oded Naor is grateful to the Technion Hiroshi Fujiwara Cyber-Security Research Center for providing a research grant. Part of Oded's work was done while at Novi Research.

*PODC '21, July 26–30, 2021, Virtual Event, Italy.*
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8548-0/21/07...$15.00
https://doi.org/10.1145/3465084.3467905

Dumbo based SMRs to run up to $n$ slots concurrently. Note, however, that for execution processes must output the slot decisions in a sequential order (no gaps). Therefore, based on the proof in [6], the time complexity of VABA and Dumbo based SMRs is $O(\log(n))$. Table 1 compares DAG-Rider to VABA and Dumbo based SMRs.

Since our protocol uses a reliable broadcast abstraction as a basic building block, different instantiations yield different complexity. For example, if we use the classic Bracha broadcast [11] to propose a single value in each message, we get a communication complexity of $O(n^3)$ per decision. This is because the Bracha broadcast complexity is $O(n^2)$, and in order to form a DAG each message has to include an $O(n)$ references to previous messages. If we are willing to allow a probability $\epsilon$ to violate progress, then we can use Guerraoui et al.'s broadcast protocol [25] and reduce the complexity to $O(n^2 \log(n))$ per decision.

Now, just as Dumbo amortizes VABA's communication complexity from quadratic to linear by using batching and adding a phase of erasure coding to more economically distribute the data, we can amortize our communication complexity to be linear per decision as well. First, since we are anyway including a vector of $O(n)$ references in every broadcast, batching $O(n)$ proposals in each message shaves a factor of $n$ of the total communication complexity even with Bracha broadcast. To arrive at the optimal linear complexity, we can replace the reliable broadcast with the asynchronous verifiable information dispersal of Cachin and Tessaro [14]. The communication complexity of that protocol is $O(n^2 \log(n) + n|V|)$, where $|V|$ is the message size, which allows us to batch $O(n \log(n))$ proposals to achieve optimal amortized communication complexity.

A final feature of our protocol, which is sometimes underestimated and cannot be presented in a table, is elegance: (1) DAG-Rider's modularity clearly separates the communication layer from the ordering logic; (2) the reliable broadcast abstraction's different instantiations yield protocols with different trade-offs, and; (3) the entire detailed pseudocode of the ordering logic spans less than 30 lines.

The rest of this paper is structured as follows: §2 describes the model and the building blocks used for DAG-Rider; §3 formally defines the BAB problem; §4 describes the DAG construction layer; §5 specifies the DAG-Rider protocol on top of the DAG layer; §6 proves the correctness of the protocol and analyzes its performance; §7 describes related work; and lastly, §8 concludes the paper.

## 2 MODEL AND BUILDING BLOCKS

The system consists of a set $\Pi = \{p_1, \ldots, p_n\}$ of $n$ processes, up to $f < n/3$ of which can act arbitrarily, i.e., be *Byzantine*. For simplicity, we consider a total of $n = 3f + 1$ processes. The link between every two correct processes is reliable. Namely, when a correct process sends a message to another correct process, the message eventually arrives and the recipient can verify the sender's identity. The communication is asynchronous, i.e., there is no bound on the message delivery time. We consider an adaptive adversary that can dynamically corrupt up to $f$ processes during the run. Once the adversary corrupts a process, it can drop undelivered messages previously sent from that process to others. The adversary controls the arrival times of messages. As part of the construction, we use two building blocks: a reliable broadcast layer and a delayed global perfect coin, which we describe next.

*Reliable broadcast.* There are known algorithms such as Bracha broadcast [11] to realize the reliable broadcast abstraction in the asynchronous network model. There are also efficient gossip protocols [9, 10, 25, 27] that provide reliable broadcast whp at a sub-quadratic communication cost in the number of processes, and asynchronous verifiable information dispersal protocols [14, 35] that use erasure codes to efficiently batch the broadcast values.

Since we are interested in constructing an asynchronous Atomic Broadcast that satisfies liveness with probability 1, we define the reliable rebroadcast abstraction accordingly to allow the use of efficient gossip protocols. Formally, each sender process $p_k$ can send 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 $r\_deliver_j(m, r, p_k)$ with probability 1.

**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 $r\_deliver_i(m, r, k)$ with probability 1.

*Global perfect coin.* We use a *global perfect coin*, which is unpredictable by the adversary. 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:

**Agreement** If two correct processes 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 $\epsilon$. 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., $\forall w \in \mathbb{N}, \forall p_j \in \Pi\colon \Pr[X_w = p_j] = 1/n$.

Such coins were used as part of previous Byzantine Agreement protocols such as [1, 7, 13, 35]. Implementation examples can be found in [13, 34]. One way to implement a global perfect coin is by using PKI and a threshold signature scheme [8, 33, 42] with a threshold of $(f + 1)$-of-$n$. When a process invokes an instance $w$ of the coin, it signs $w$ with its private key and sends the share to all the processes. Once a process receives $f + 1$ shares, it can combine them to get the threshold signature and hash it to get a random process.

| | Communication Complexity | Expected time Complexity | Post-Quantum Safety | Eventual Fairness |
|---|---|---|---|---|
| VABA SMR | $O(n^2)$ | $O(\log(n))$ | no | no |
| Dumbo SMR | amortized $O(n)$ | $O(\log(n))$ | no | no |
| DAG-Rider + [11] | amortized $O(n^2)$ | $O(1)$ | yes | yes |
| DAG-Rider + [25] | amortized $O(n\log(n))$ | $O\left(\frac{\log(n)}{\log(\log(n))}\right)$ | yes | (1-$\epsilon$)-fair |
| DAG-Rider + [14] | amortized O(n) | $O(1)$ | yes | yes |

**Table 1: A comparison between our protocol with different reliable broadcast instantiations and VABA and Dumbo based SMR protocols. Eventual fairness means that all proposals by correct processes are eventually ordered.**

Since the threshold signature value is deterministically determined by the instance name $w$ such that any $f+1$ shares reveal it (e.g., the schema in [42] is based on Shamir's secret sharing [41]), the coin is perfect (all process agree on the leader) and its agreement property has information theoretical guarantee. However, to ensure unpredictability, the PKI must be trusted to ensure that the adversary cannot generate enough shares to reveal the randomness before a correct process produces them. Usually, one assumes that a trusted dealer is used to set up the random keys for all processes. However, this assumption can be relaxed by executing an $O(n^4)$ message complexity Asynchronous Distributed Key Generation protocol [30]. Either way, this scheme remains unpredictable only if the adversary is computationally bounded. However, since DAG-Rider relies on the unpredictability property of the coin only for liveness, its safety properties are post-quantum secure.

## 3 PROBLEM DEFINITION

The problem we solve is *Byzantine Atomic Broadcast (BAB)*, which allows processes to agree on a sequence of messages as needed for State Machine Replication (SMR). Due to the FLP result [23], BAB cannot be solved deterministically in the asynchronous setting, and therefore we use the global perfect coin to provide randomness that ensures liveness with probability 1. To avoid confusion with the events of the underlying reliable broadcast abstraction, we name the broadcast and deliver events of BAB as $a\_bcast(m, r)$ 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 process. 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 3.1** (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 $a\_deliver_j(m, r, k)$.

In the context of Byzantine SMR (e.g., blockchains), the BAB abstraction support the separation between sequencing of transactions and execution as done in [2]. BAB provides a mechanism to propose transactions and totally order them, and an execution engine will have to validate the transactions before applying them to the SMR.

Moreover, note that our BAB definition provides a stronger guarantee than the one provided by the sequencing protocols realized in most Byzantine SMR systems. Our validity property requires that all messages broadcast by correct processes are eventually ordered (with probability 1), whereas most Byzantine SMR protocols (i.g., [17, 37, 43] require that in an infinite run, an infinite number of decisions are made, but some proposals by correct processes can be ignored. In addition, it is important to note that our BAB protocol satisfies chain quality. That is, for every prefix of ordered messages of size $(2f+1)r$, $r \in \mathbb{N}$, at least $(f+1)r$ were broadcast by correct processes.

*Communication measurement.* To analyze amortized communication complexity we assume that each message contains a *block* of transactions, and we say that a transaction in a message $m$ is *ordered* when all honest parties $a\_deliver\ m$. We measure *communication complexity* as the total number of bits sent by honest processes to order a single transaction. To be able to measure the asynchronous running time we follow [16] and define a *time unit* for every execution $r$ to be the maximum time delay of all messages among correct processes in $r$. We measure *time complexity* as the expected number of time units it takes for a correct party to deliver $O(n)$ values proposed by different correct processes starting from any point in the execution.

## 4 DAG ABSTRACTION

Our BAB protocol, DAG-Rider, is based on a Directed Acyclic Graph (DAG) abstraction, which represents the communication layer of the processes. In a nutshell, each vertex in the DAG represents a reliable broadcast message from a process, and each message contains, among other data, references to previously broadcast vertices. Those references are the edges of the DAG. Each correct process maintains a copy of the DAG as it perceives it. Different correct processes might observe different states of the DAG during different times of the run, but reliable broadcast prevents equivocation and guarantees that all correct processes eventually deliver the same messages, so their views of the DAG eventually converge.

For each process $p_i$, denote $p_i$'s local view of the DAG as $DAG_i$, which is stored as an array $DAG_i[]$. As we shortly explain, each vertex in the DAG is associated with a unique round number and a source (its generating process). At any given time, $DAG_i[r]$ for $r \in \mathbb{N}$ is the set of all the vertices associated with round $r$ that $p_i$ is

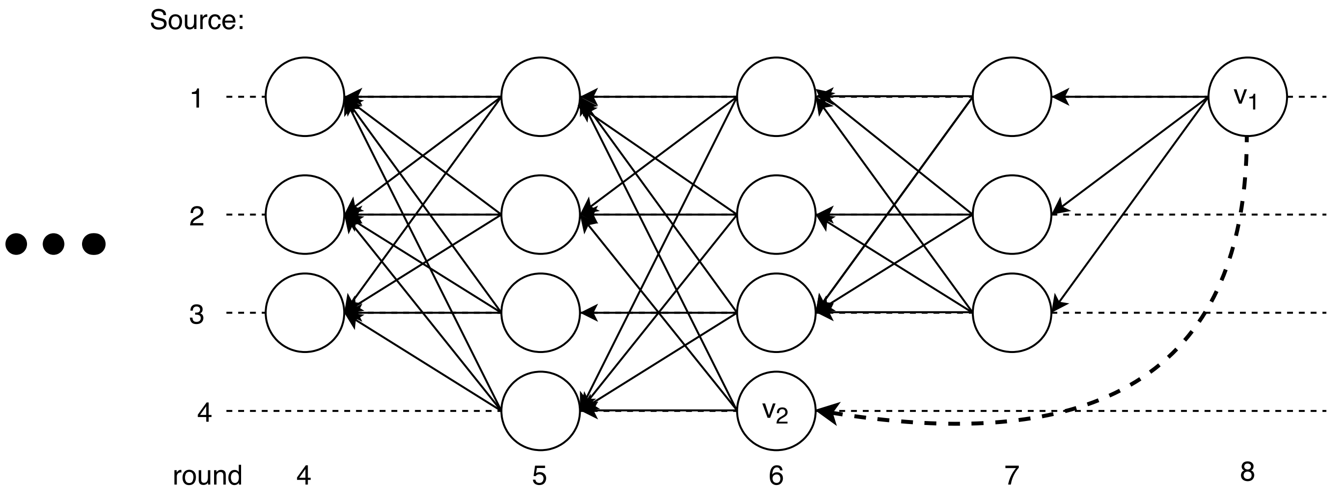

**Figure 1: Illustration of DAG$_1$, i.e., the DAG at process 1, out of a total of four processes. On each horizontal dotted line are the vertices from a single source, e.g, the bottom line shows the vertices delivered from process 4. Each vertical column of vertices is a single round. Each completed round has at least $2f+1=3$ vertices. Each vertex in the DAG has at least $2f+1$ strong edges to vertices from the previous round shown as black solid arrows. Each vertex can also have weak edges to vertices in case there is no other path in the DAG to the vertex. E.g., $v_1$ in the illustration has a weak edge to $v_2$, shown as a dotted arrow to $v_2$.**

**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 process 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
    $DAG[]$ - An array of sets of vertices, initially:
        $DAG_i[0] \leftarrow$ predefined hardcoded set of $2f+1$ 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 $\forall i \in [2..k]: v_i \in \bigcup_{r \geq 1} DAG_i[r] \wedge v_i \in v_{i-1}.strongEdges$
```

aware of. Each round has at most $n$ vertices, each with a different source. Due to the reliable broadcast, no process can generate two vertices in the same round.

Each vertex $v$ in a round $r$ has two sets of outgoing edges: a set of at least $2f+1$ *strong edges* and a set of up to $f$ *weak edges*. Strong edges point to vertices in round $r-1$ and weak edges point to vertices in rounds $r' < r-1$ such that otherwise there is no path from $v$ to them. As explained in detail in §5, strong edges are used for agreement and weak edges make sure we eventually include all vertices in the total order, to satisfy BAB's validity property.

The data types and variables for process $p_i$ are specified in Algorithm 1 and the DAG construction is specified in Algorithm 2. A vertex $v$ is a struct that holds a round number $r$, a source which is the process that created $v$, a block of valid transactions that was previously *a_bcast* by the upper BAB protocol, strong edges to at least $2f+1$ vertices in round $r-1$, and weak edges to vertices in rounds $r' < r-1$. Vertices in the DAG are reliably broadcast (Line 15), and when the reliable broadcast layer delivers a vertex $v$ (Line 22), processes use the round number $r$ and the source process which are available from the reliable broadcast and add them to $v$. Then, they verify that $v$ has strong edges to at least $2f+1$ vertices from round $r-1$ and add it to a buffer.

Each process $p_i$ continuously goes through its buffer to check if there is a vertex $v$ in it that can be added to its $DAG_i$ (Line 6). A vertex $v$ can be added to the DAG once the DAG contains all the vertices that $v$ has a strong or weak edge to (Line 7). When $p_i$ has

**Algorithm 2 DAG Construction**, pseudocode for process $p_i$

```
    Local variables:
        $r \leftarrow 0$                                                  ▷ round number
        $buffer \leftarrow \{\}$
5:  while True do
6:      for $v \in buffer$: $v.round \le r$ do
7:          if $\forall v' \in v.strongEdges \cup v.weakEdges$: $v' \in \bigcup_{k \ge 1} DAG[k]$ then    ▷ We have $v$'s predecessors
8:              $DAG[v.round] \leftarrow DAG[v.round] \cup \{v\}$
9:              $buffer \leftarrow buffer \setminus \{v\}$
10:     if $|DAG[r]| \ge 2f + 1$ then                                     ▷ Start a new round
11:         if $r \bmod 4 = 0$ then                                       ▷ If a new wave is complete
12:             $wave\_ready(r/4)$                  ▷ Signal to Algorithm 3 that a new wave is complete
13:         $r \leftarrow r + 1$
14:         $v \leftarrow create\_new\_vertex(r)$
15:         $r\_bcast_i(v, r)$
16: procedure $create\_new\_vertex$(round)
17:     wait until $\neg blocksToPropose$.empty()      ▷ atomic broadcast blocks are enqueued (Line 32)
18:     $v.block \leftarrow blocksToPropose$.dequeue()   ▷ We assume each process atomically broadcast infinitely many blocks
19:     $v.strongEdges \leftarrow DAG[round - 1]$
20:     $set\_weak\_edges(v, round)$
21:     return $v$
22: upon $r\_deliver_i(v, round, p_k)$ do               ▷ The deliver output from the reliable broadcast
23:     $v.source \leftarrow p_k$
24:     $v.round \leftarrow round$
25:     if $|v.strongEdges| \ge 2f + 1$ then
26:         $buffer \leftarrow buffer \cup \{v\}$
27: procedure $set\_weak\_edges(v, round)$               ▷ Add weak edges to orphan vertices
28:     $v.weakEdges \leftarrow \{\}$
29:     for $r = round - 2$ down to 1 do
30:         for every $u \in DAG_i[r]$ s.t. $\neg path(v, u)$ do
31:             $v.weakEdges \leftarrow v.weakEdges \cup \{u\}$
```

at least $2f + 1$ vertices in the current round, it moves to the next round (Line 10) by creating and reliably broadcasting a new vertex $v'$. The new vertex $v'$ in round $r$ includes a block of transactions $b$ for which $p_i$ previously invoked $a\_bcast(b, r)$ (we assume each process atomically broadcast infinitely many blocks), strong edges to the vertices in $DAG_i[r]$ (Line 16), and weak edges to any vertices with no path from $v'$ to them (Line 27). Note that a vertex might be delivered at $p_i$'s DAG after $p_i$ has moved to a later round. In this case, the vertex is still added to the DAG, but $p_i$'s vertices do not include strong edges to it. Weak edges are possible. As noted, the weak edges are used to ensure the BAB's Validity property. An example of our DAG construction is illustrated in Fig. 1.

## 5 DAG-RIDER: DAG-BASED ASYNCHRONOUS BAB PROTOCOL

In this section, we describe the DAG-Rider protocol, by equipping the DAG from the previous section with a global perfect coin[1] and show how the DAG and the coin can be used to construct a locally-computed protocol for the BAB problem. That is, given our DAG and a perfect coin, DAG-Rider does not require any extra communication among the processes. Instead, each process $p_i$ observes its local $DAG_i$ and deduces which blocks of transactions to deliver and in what order. The protocol is detailed in Algorithm 3. Below we give a high-level intuition as well as a detailed description of the protocol. Formal correctness proofs and complexity analyzes are given in §6.

When an $a\_bcast_i(b, r)$ is invoked, $p_i$ simply pushes $b$ to the DAG layer (line 33), which in turn includes it in the $r^{th}$ vertex it reliably broadcasts. To interpret the DAG, each process $p_i$ divides its local $DAG_i$ into waves, where each wave consists of 4 consecutive rounds. For example, $p_i$'s first wave consists of $DAG_i[1], DAG_i[2], DAG_i[3]$, and $DAG_i[4]$. Formally, the $k^{th}$ round of wave $w$, where $k \in [1..4], w \in \mathbb{N}$, is defined as $round(w, k) \triangleq 4(w - 1) + k$. We also say that a process $p_i$ *completes round* $r$ once $DAG_i[r]$ has at least $2f + 1$ vertices, and a process *completes wave* $w$ once the process completes $round(w, 4)$.

In a nutshell, the idea is to interpret the DAG as a wave-by-wave protocol and try to *commit* a randomly chosen single leader vertex in every wave. Once the sequence of leaders is determined, processes $a\_deliver$ all the blocks included in their causal histories (in vertices that have paths from the leaders in the DAG). While reading the high-level description below, bear in mind that due to the reliable broadcast, Byzantine processes cannot equivocate, so two correct processes cannot have different vertices with the same

[1] A possible implementation of the coin using threshold signatures is described in §2. The coin can be easily implemented as part of the DAG itself by having each process send its share of the threshold signature when reliably broadcasting a vertex.

**Algorithm 3 DAG-Rider: Byzantine Atomic Broadcast based on DAG.** Pseudocode for process $p_i$

```
    Local Variables:
        decidedWave ← 0
        deliveredVertices ← {}
        leadersStack ← initialize empty stack with isEmpty(), push(), and pop() functions
32: upon $a\_bcast_i(b, r)$ do            ▷ Correct processes call this procedure with sequential round $r$ numbers, starting at 1
33:     $blocksToPropose$.enqueue($b$)                                       ▷ pushes a block of transactions to Alg 2
34: upon $wave\_ready(w)$ do                    ▷ Signal from the DAG layer that a new wave is completed (Line 12)
35:     $v \leftarrow get\_wave\_vertex\_leader(w)$
36:     if $v = \bot \lor |\{v' \in DAG_i[round(w, 4)] : strong\_path(v', v)\}| < 2f + 1$ then                ▷ No commit
37:         return
38:     $leadersStack$.push($v$)
39:     for wave $w'$ from $w - 1$ down to $decidedWave + 1$ do
40:         $v' \leftarrow get\_wave\_vertex\_leader(w')$
41:         if $v' \neq \bot \land strong\_path(v, v')$ then
42:             $leadersStack$.push($v'$)
43:             $v \leftarrow v'$
44:     $decidedWave \leftarrow w$
45:     $order\_vertices(leadersStack)$
46: procedure $get\_wave\_vertex\_leader(w)$
47:     $p_j \leftarrow choose\_leader_i(w)$
48:     if $\exists v \in DAG[round(w, 1)]$ s.t. $v.source = p_j$ then
49:         return $v$                                                ▷ There can only be one such vertex
50:     return $\bot$
51: procedure $order\_vertices(leadersStack)$
52:     while $\neg leadersStack$.isEmpty() do
53:         $v \leftarrow leadersStack$.pop()
54:         $verticesToDeliver \leftarrow \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v, v') \land v' \notin deliveredVertices\}$
55:         for every $v' \in verticesToDeliver$ in some deterministic order do
56:             output $a\_deliver_i(v'.block, v'.round, v'.source)$
57:             $deliveredVertices \leftarrow deliveredVertices \cup \{v'\}$
```

source in the same round, leading to eventually consistent DAGs among all correct processes.

When wave $w$ completes (Line 34), we use the global perfect coin to retrospectively elect some process and consider its vertex in the wave's first round as the leader of wave $w$ (Line 35). The goal of the protocol is to commit this leader, provided that it has been observed by sufficiently many processes in the wave. Note that since we advance rounds as soon as we deliver $2f + 1$ of the $3f + 1$ potential vertices, a process $p_i$ might not have $w$'s leader in its local $DAG_i$ when it completes $w$. In this case, $p_i$ completes $w$ without committing any vertex and simply proceeds to the next wave. Note, however, that some other correct process might have $w$'s leader in its local DAG and commit it in the same wave. Thus, we need to make sure that if one correct process commits the wave vertex leader $v$, then all the other correct processes will commit $v$ later. To this end, we use standard quorum intersection. Process $p_i$ commits the wave $w$ vertex leader $v$ if:

$$\left|\left\{v' \in DAG_i[round(w, 4)] : strong\_path(v', v)\right\}\right| \geq 2f + 1 \text{ (Line 36).}$$

In addition, if $p_i$ commits vertex $v$ in wave $w$ and there is a strong path from $v$ to $v'$ such that $v'$ is an uncommitted leader vertex in a wave $w' < w$, then $p_i$ commits $v'$ in $w$ as well. The leaders committed in the same wave are ordered by their round numbers, so that leaders of earlier waves are ordered before those of later ones, meaning $v'$ is ordered before $v$ (Lines 39-43).

The next lemma, which is proven in Section 6, shows that our commit rule guarantees that if a correct process commits a wave leader vertex $v$ in some wave, then all wave vertex leaders in later waves in the local DAGs of all correct processes have a strong path to $v$, ensuring the agreement property.

**Lemma 1.** *If some process $p_i$ commits the leader vertex $v$ of a wave $w$, then for every leader vertex $u$ of a wave $w' > w$ and for every process $p_j$, if $u \in DAG_j[round(w', 1)]$, then $strong\_path(u, v)$ returns true in wave $w'$.*

We show below how we leverage the above lemma to satisfy the total order property, but first, we give an intuition for liveness, i.e., the validity and agreement properties. Our protocol achieves progress in a constant number of waves, in expectation, by guaranteeing that for every wave, the probability for every correct process to commit the wave leader is at least $2/3$. To ensure this, we borrow the technique from the *common-core abstraction* [15], which guarantees that after three rounds of all-to-all sending and collecting accumulated sets of values, all correct processes have at least $2f + 1$ common values. The set of these values is referred to as the common-core. In respect to our DAG, we prove in Section 6 the following lemma:

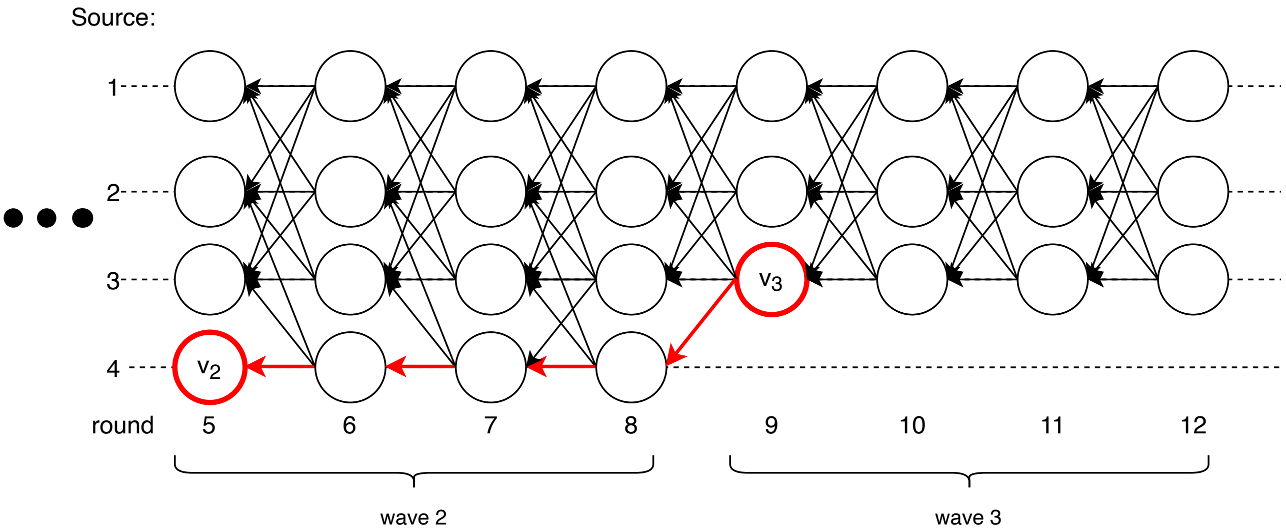

**Figure 2: Illustration of $DAG_1$. The highlighted vertices $v_2$ and $v_3$ are the leaders of waves 2 and 3, respectively. The commit rule is not met in wave 2 since there are less than $2f + 1$ vertices in round 8 with a strong path to $v_2$. However, the commit rule is met in wave 3 since there are $2f + 1$ vertices in round 12 with a strong path to $v_3$. Since there is a strong path from $v_3$ to $v_2$ (highlighted), $p_i$ commits $v_2$ before $v_3$ in wave 3.**

**Lemma 2.** *Let $p_i$ be a correct process that completes wave $w$. Then there is a set $V \subseteq DAG_i[round(w, 1)]$ and a set $U \subseteq DAG_i[round(w, 4)]$ s.t. $|V| \geq 2f + 1$, $|U| \geq 2f + 1$ and $\forall v \in V, \forall u \in U$: $strong\_path(u, v)$.*

Note that by the commit rule, if the leader of a wave $w$ belongs to the set $V$ (from the lemma statement), then $p_i$ commits the leader once it completes $w$. So to deal with an adversary that totally controls the network, parties flip the global coin only after they complete $w$ (Line 35). Therefore, by the coin's unpredictability property, the probability of the adversary to guess the wave's leader before the point after which it cannot manipulate the set $V$ is less than $\frac{1}{n} + \epsilon$. Thus, with a probability of at least $2/3 - \epsilon$, $w$'s leader is in the set $V$ and $p_i$ commits it. Thus, in expectation, correct processes commit every 3/2 waves.

To satisfy total order, we leverage the property proven in Lemma 1 to make sure all processes commit the same waves' leaders. Once we find a leader to commit in a wave $w$ we check if it is possible that some process committed a wave in between $w$ and the previous wave we committed, let it be $w'$. We do this iteratively in Lines 39-43, we first check if it is possible that some process committed the leader of $w - 1$. We do it by checking if there is a strong path from the leader of wave $w$ to the leader of wave $w - 1$ in our local DAG (Line 41). If no such path exists, by Lemma 1, no correct process will ever commit $w - 1$. Otherwise, we choose to commit $w - 1$ before $w$. Now, if such a path indeed exists, we recursively check if it is possible that some process committed a wave in between $w - 1$ and $w'$. Otherwise, if no such path exists, we check if there is a path from the leader of wave $w$ to the leader of wave $w - 2$ and continue in the same way. The recursion ends once we reach a wave that we previously committed, $w'$ in our example. An illustration of this process is given in Fig. 2.

Since vertices are reliably broadcast and since we never add a vertex $v$ to the DAG before we add all the vertices $v$ points to with strong or weak edges, two correct processes always have the same causal history for any vertex they both have in their DAGs. Therefore, once we agree on a sequence of leaders, all that is left to do is to order the causal histories of the leaders in some deterministic order. To this end, we go through the waves' committed leaders one-by-one and *a_deliver*, in some deterministic order, all the transaction blocks in their causal histories that we did not previously deliver (procedure *order_vertices* in Line 51). The causal history of a wave leader vertex $v$ in $DAG_i$ is the set $\{u \in DAG_i \mid path(v, u)\}$.

The purpose of the weak edges is to satisfy the Validity property. Recall that strong edges might not point to all vertices from the previous round in the DAG because we might advance the round before we deliver all the broadcasts of that round (we advance the round once at least $2f + 1$ vertices are added to the DAG). Therefore, without the weak edges, slow processes may not be able to get vertices from higher rounds to point to theirs. So to satisfy Validity, each correct process, when creating a new vertex, adds weak edges to all vertices in its local DAG to which it otherwise does not point.

## 6 ANALYSIS

In §6.1 we prove the correctness of DAG-Rider, and in §6.2 we analyze the communication and time complexity.

### 6.1 Correctness

We show that DAG-Rider satisfies the properties of the BAB problem, as defined in §3.

Proposition 1. *DAG-Rider satisfies the integrity property.*

Proof. By the code (Line 56), if a correct process $p_i$ outputs $a\_deliver(b, r, p_k)$, then there is a vertex $v$ in $DAG_i$ s.t. $(b, r, p_k) = (v.block, v.round, v.source)$. Integrity follows from the fact that all vertices are reliably broadcast, and thus by integrity property of the reliable broadcast there are no two different vertices $u, u'$ in $DAG_i$ s.t. $u.round = u'.round$ and $u.source = u'.source$. □

**Claim 1.** *When a correct process $p_i$ adds a vertex $v$ to its $DAG_i$ (Line 8), all of $v$'s causal history is already in $DAG_i$.*

Proof. We prove this claim by induction on the execution of every correct process $p_i$. Denote by $v_k$ the $k$-th vertex that $p_i$ adds to $DAG_i$. We show that for every $k \in \mathbb{N}$, after $v_k$ is added to the DAG, the causal histories of all the vertices in the set $\{v_1, \ldots, v_k\}$, and in particular $v_k$, are in $DAG_i$.

In the base step of the induction, there are no vertices in the DAG, and the property vacuously holds. Next, assume that after $v_k$ is added to the DAG at process $p_i$, all the causal histories of all the vertices in the set $V = \{v_1, \ldots, v_k\}$ are already in $DAG_i$.

For $v_{k+1}$ to be added to the DAG at process $p_i$, its strong and weak edges must reference vertices that are already in $DAG_i$ (Line 7), i.e., $v_{k+1}$'s edges are only to vertices in $V$. Since all the vertices in $V$ already have their causal histories in the DAG, when $v_{k+1}$ is added to the DAG, its causal history is in the DAG as well, and we are done. □

**Claim 2.** *If a correct process $p_i$ adds a vertex $v$ to its $DAG_i$, then eventually all correct processes add $v$ to their DAG.*

Proof. By induction on rounds, for process $p_i$ to add a vertex $v$ in round $r$ to its $DAG_i$, first $v$ needs to be delivered to $p_i$ by the reliable broadcast layer (Line 22), and by the agreement of the reliable broadcast, $v$ will be eventually delivered to all other correct processes.

Next, $v$ has to be added to the *buffer* variable at $p_i$, and this is done if the process who broadcast $v$ added the correct $v.source$ and $v.round$ which are verified through the guarantees of the reliable broadcast layer (Line 25). Therefore these checks will also pass at any other correct process when $v$ is delivered to it. Finally, $p_i$ checks that the vertex has at least $2f + 1$ strong edges to vertices in round $v.round - 1$. If $v$ passes this check in $p_i$ then it will pass these two checks at any other correct process, since these checks are computed locally based on $v$'s fields ($v.block$ and $v.strongEdges$).

Lastly, after $v$ is added to the *buffer*, for $p_i$ to add $v$ to its $DAG_i$, $p_i$ also checks that it has all the vertices that $v$ is referencing to (in $V = v.strongEdges \cup v.weakEdges$) in its $DAG_i$ as well (Line 7). By the induction assumption, all correct processes' DAGs contain the same vertices in rounds $< r$.

Thus, this ensures that any vertex $v$ that appears in any round at $DAG_i$ of some correct process, will eventually also appear in $DAG_j$ of every other correct process $p_j$. □

**Claim 3.** *If for some correct process $p_i$ there is a round $r$ with a set $V$ of at least $2f + 1$ vertices in $DAG_i[r]$ s.t. $\forall v \in V$: $strong\_path(v, u)$ to some vertex $u \in DAG_i$, then every other process $p_j$ that completes round $r$ has a set $V' \subseteq DAG_j[r]$ s.t. $|V'| \geq f + 1$ and $\forall v' \in V'$: $strong\_path(v', u)$.*

Proof. Let $V' = V \cap DAG_j[r]$. Round $r$ is complete for $p_i$ and $p_j$ when their DAGs have at least $2f + 1$ vertices. Therefore, when $p_i$ and $p_j$ complete round $r$, $|V'| \geq f + 1$ by a standard quorum intersection of $2f + 1$ out of $3f + 1$ possible vertices of round $r$ (due to the reliable broadcast, Byzantine processes cannot equivocate). Since every $v' \in V'$ is already in $DAG_j$ when $p_j$ completes round $r$, then $u$ is in $DAG_j$ by $t$ as well (by Claim 1), and there is a strong path between every $v' \in V'$ to $u$ in $DAG_j$. □

For the next part, we say a vertex $v$ is a wave $w$ vertex leader if $v$ is the return value of the *get_wave_vertex_leader* procedure (Line 46). Next, we say a process *commits* a wave leader vertex $v$ when $v$ is popped from the stack (Line 53).

**Claim 4.** *In every wave, at most one vertex $v$ can be a wave leader vertex for all correct processes.*

Proof. For a vertex $v$ to be a wave leader vertex in wave $w$ it has to be the return value from the *get_wave_vertex_leader* procedure (Line 46). The procedure gets the wave's chosen process $p_j$ by the global coin, and checks if the $DAG_i$ at process $p_i$ has the vertex $v$ from $p_j$ in the first round of wave $w$. Due to the agreement property of the global perfect coin, the same process $p_j$ is chosen for all correct processes, and because of the agreement property of the reliable broadcast, Byzantine processes cannot equivocate. □

**Claim 5.** *If a correct process $p_i$ commits wave leader vertex $v_1$ in wave $w_1$ and then $p_i$ commits vertex $v_2$ in wave $w_2$, then $w_1 < w_2$.*

Proof. A vertex is committed when it is popped from the stack (Line 53). Vertices are pushed to the stack in Lines 38 and 42, which only happens in waves which vertices were not committed before, since the for loop goes down only to *decidedWave* + 1 (Line 39), where *decidedWave* is updated each time vertices are pushed to the stack to be the maximum wave in which vertices were committed (Line 44). This means that vertices are pushed to the stack in decreasing wave numbers.

Lastly, all the vertices in the stack are popped out and committed, and this is done in reverse order to the order that they were pushed to the stack, therefore, the wave numbers of committed waves are in an increasing order. □

**Lemma 1.** *If some process $p_i$ commits the leader vertex $v$ of a wave $w$, then for every leader vertex $u$ of a wave $w' > w$ and for every process $p_j$, if $u \in DAG_j[round(w', 1)]$, then $strong\_path(u, v)$ returns true in wave $w'$.*

Proof. Since vertex $v$ is committed by process $p_i$ in wave $w$, the commit rule is met, i.e., at the end of wave $w$ there are at least $2f + 1$ vertices in $DAG_i[round(w, 4)]$ with a strong path to $v$. By Claim 3, every correct process $p_j$ (whether it committs $v$ in $w$ or not) has a set $V$ of at least $f + 1$ vertices in $DAG_j[round(w, 4)]$ with a strong path to $v$. Any future vertex $v'$ from waves $w' > w$, including $u$, will have a strong path to at least one vertex in $V$, resulting in a strong path between $u$ and $v$. □

Proposition 2. *DAG-Rider satisfies the total order property.*

Proof. By Claim 4, each wave has only one vertex that can be committed. By Claim 5 every correct process commits vertices in an increasing wave number. By Lemma 1, if a correct process

$p_i$ commits a vertex $v$, then there is a strong path to $v$ from any vertex $u$ in future waves that might be committed. By combining all the claims, if two correct processes commit the same wave leader vertices, they do so in the same order.

Once a correct process commits a wave vertex leader $v$, it atomically delivers all of $v$'s causal history in some deterministic order, which is identical for all other correct processes. By Claim 1, when $v$ is committed, all of $v$'s causal history is already in the DAG. Thus, since all correct processes commit the same wave leader vertices in the same order, and since those vertices have the same causal histories, all correct processes that deliver vertices, do so in the same order. □

**Lemma 2.** *Let $p_i$ be a correct process that completes wave $w$. Then there is a set $V \subseteq DAG_i[round(w,1)]$ and a set $U \subseteq DAG_i[round(w,4)]$ s.t. $|V| \geq 2f+1$, $|U| \geq 2f+1$ and $\forall v \in V, \forall u \in U: strong\_path(u,v)$.*

Proof. First, we show that there is a set $V$, $|V| \geq 2f+1$ s.t. when $p_i$ completes $round(w,3)$ and broadcasts a new vertex $v_4$ in $round(w,4)$, then $v_4$ has a strong path to all the vertices in $V$.

To this end, we use the common-core abstraction, that first appeared in [15], and was adapted (and proven) for the Byzantine case in [20]. The model for this abstraction is identical to our model. Each correct process $p_i$ has some input value $v_i$, and it returns a set $V_i$ of input values from different processes. The guarantee of the common-core abstraction is that there is a subset $V$ of at least $2f+1$ values, s.t. for each correct process $V \subseteq V_i$, i.e., there is a common core of at least $2f+1$ input values that appear in the returned sets of all the correct processes that complete the common-core abstraction.

The algorithm to realize the common-core abstraction consists of three rounds of communication: in the first round, each process sends its input value $v_i$, and then waits for $2f+1$ input values from other processes (including itself). Denote this first set at process $p_i$ as $F_i$.

In the second stage, each process sends its $F_i$ set and waits until it receives $2f+1$ $F_j$ sets from other processes (including itself). When this stage ends, process $p_i$ creates the union of all the $F_j$ sets it received. Denote this set of sets for process $p_i$ as $S_i$. In the third and last stage, process $p_i$ sends the set $S_i$ it created and again waits to receive $2f+1$ $S_j$ sets from other processes (including itself). When this stage ends, process $p_i$ returns the union of all the $S_j$ sets, denoted $T_i$, as the output of the common-core abstraction.

We show that the first three rounds of a wave $w$ can be mapped exactly to the three stages of the common-core algorithm. Denote $r_1, r_2, r_3, r_4$ as $round(w,1)$, $round(w,2)$, $round(w,3)$, $round(w,4)$, respectively. When a correct process $p_i$ adds the vertex $v_1$ created in $r_1$ to $DAG_i[r_1]$, by Claim 2, eventually all other correct processes add $v_1$ to their DAG, which can be mapped to $p_i$ sending its input value to all other processes in the common-core algorithm. Next, $p_i$ moves to round $r_2$ once it has at least $2f+1$ vertices in $r_1$, which is mapped to $p_i$ waiting for $2f+1$ input values from different processes in the common-core algorithm. When $p_i$ enters $r_2$ it broadcasts a vertex $v_2$ that references all the vertices it has in $r_1$, which is equivalent to $p_i$ sending $F_i$ at the beginning of the second stage of the common-core algorithm. In a similar way, when $p_i$ completes $r_2$ and enters $r_3$, it broadcasts $v_3$ which references all the vertices it has in $r_2$, which is equivalent to sending $S_i$ (by Claim 1, when $v_3$ is added to $DAG_j[r_3]$ of some correct process $p_j$, then all the vertices $p_i$ has in $DAG_i[r_1]$ with a strong path from $v_3$ are in the $DAG_j[r_1]$ as well). To complete the mapping, when $p_i$ completes $r_3$ and broadcasts $v_4$ in round $r_4$, then $v_4$ has in its causal history the same values that would have been in $T_i$ in the equivalent common-core algorithm.

Note that since Byzantine processes cannot equivocate, and since every round in the DAG has at least $2f+1$ vertices, any vertex that $p_i$ adds to $DAG_i[r_4]$ has to reference at least $f+1$ vertices that $v_4$ also references, even vertices sent from Byzantine processes. Thus, based on the common-core guarantee, there is a set $V \subset DAG_i[r_1]$ s.t. $|V| \geq 2f+1$ and $\forall v \in V: strong\_path(v_4, v)$, and also this set $V$ appears in the DAG of any other correct process $p_j$ that completes round $r_3$. Next, when $p_i$ completes wave $w$, i.e., when it completes round $r_4$, it has in $DAG_i[r_4]$ at least $2f+1$ vertices, and each of those vertices has a path to each of the vertices in $V$, which concludes the proof. □

**Claim 6.** *For every correct process $p_i$ and for every wave $w$, the expected number of waves, starting from $w$, until the commit rule is met is equal to or smaller than $3/2+\epsilon$.*

Proof. By Lemma 2, in each wave $w$, the probability that for a correct process $p_i$ the commit rule is met is at least $pr = (2f+1)/(3f+1) - \epsilon$. The number of waves until the commit rule is met is geometrically distributed with a success probability of $pr$. Thus, the expected number of waves is bounded by $3/2+\epsilon$ waves. □

Proposition 3. *DAG-Rider guarantees the agreement property.*

Proof. If a correct process $p_i$ outputs $a\_deliver_i(b, r, p_k)$ it means that $b$ is a block of some vertex $u$ that is in the causal history of some wave's $w$ leader vertex $v$ that, i.e., when process $p_i$ commits a wave vertex leader $v$, then $u$ is in $v$'s causal history.

By Claim 6, every other correct process $p_j$ that has not committed $v$ yet will eventually, with probability 1, have a wave $w' > w$ in which the commit rule is met. When $p_j$ commits $w'$, by the proved total order property, it will also commit $v$, and thus decide on all of $v$'s causal history in the same order, including vertex $u$. □

**Claim 7.** *Every vertex that is broadcast by a correct process is eventually added to the DAG of all correct processes.*

Proof. We prove this by showing that for every correct process $p_i$ that broadcasts a vertex $v$, $v$ is eventually added to $DAG_i$, and by Claim 2, $v$ is eventually added to the DAG of all other correct processes.

When a correct process $p_i$ broadcasts a vertex $v$ (Line 15) it broadcasts a valid vertex, i.e., a vertex that passes the external validity check, and that references vertices that are already in $DAG_i$. Because of the validity property of the reliable broadcast, $o_i$ eventually delivers $v$ to itself, and when it does so, it adds $v$ to its own $DAG_i$. Thus, as explained, by Claim 2, $v$ is eventually added to the DAGs of all other correct processes. □

Proposition 4. *DAG-Rider guarantees the validity property.*

Proof. When a correct process $p_i$ calls *a_deliver* with some value, it is inserted into a queue (Line 33), and eventually will be included in a vertex $v$ created by $p_i$ (Line 18). Vertex $v$ is eventually reliably delivered to all the correct processes and added to their DAGs (Claim 7).

When a correct process reliably broadcasts a new vertex $v$ in round $r$ it also makes sure that it has a path (either a strong or weak path) to all the vertices in rounds $r' < r$, and if not, it adds weak edges to $v$ that guarantee this (Line 27), therefore $v$ will eventually be included in the causal history of all correct processes. Eventually, with probability 1, $v$ will be in the causal history of a committed wave vertex leader, and therefore atomically delivered. □

### 6.2 Communication and Time Complexity

We analyze DAG-Rider in terms of expected communication complexity and expected time complexity.

*Communication complexity.* We analyze the communication complexity of DAG-Rider when instantiated with Cachin and Tesero's [14] information dispersal protocol. A similar analysis can be made for other broadcast implementations as well. For clarity, in §4, we say that strongEdges and weakEdges are sets of vertices. However, in order to refer to a vertex it is enough to only store its source and round number.[2] We assume that any round number during an execution can be expressed in a constant number of bits, that is, the DAG never reaches round number $2^{128}$ (note that round numbers grow slower than slot numbers).

We count the number of bits sent by correct processes in every round of the DAG and divide it by the total number of ordered values therein. The complexity of [14] is $O(n^2 \log(n) + nM)$, where $M$ is the message (vertex) size. Each message includes a set of proposed values and $n$ references, and each reference includes a process id of size $\log(n)$. Thus, if we batch $n \log(n)$ values in every message, the bit complexity is $O(n^2 \log(n) + 2n^2 \log(n)) = O(n^2 \log(n))$ for a broadcast.

Since each process is allowed to broadcast a single message in each round, a correct process will not participate in more than $n$ reliable broadcasts in a round, and thus the total bit complexity of correct processes in a round is bounded by $O(n^3 \log(n))$. On the other hand, at least $2f + 1 = O(n)$ vertices are ordered in every round. Thus, $O(n^2 \log(n))$ values are ordered in every round. Hence, DAG-Rider's amortized communication complexity of is $O(n)$.

*Time complexity.* By Claim 6, the number of waves, in expectation, between two waves that satisfy the commit rule in $DAG_i$ for a correct process $p_i$ is expected constant. Since each wave consists of constant size chains of messages, by the definition of time units, the number of time units, in expectation, between two $p_i$'s commits is constant. Every time $p_i$ commits a wave, it commits the wave's leader causal history, which contains at least $O(n)$ proposals from different correct processes. Therefore, DAG-Rider's time complexity is $O(1)$ in expectation.

## 7 RELATED WORK

The first asynchronous Byzantine Agreement protocols [5, 39] showed that the FLP [23] impossibility can be circumvented with randomization. Their communication and time complexity was exponential and a significant amount of work has been done since then in attempt to achieve optimal complexity under different assumptions. Some works consider the information theoretical settings and present protocols with polylogarithmic complexity that tolerate adversaries with unbounded computational power [4, 26, 38]. Others follow a more practical approach and consider a computationally bounded adversary to be able to use cryptographic primitives to improve complexity [1, 12, 13, 35]. The pioneering crypto-based protocols [12, 13] were later realized in HoneyBadgerBFT, the first asynchronous Byzantine SMR system [36]. However, while the state-of-the-art protocols VABA [1] and Dumbo [35] rely on cryptographic assumptions for both safety and liveness, DAG-Rider uses a hybrid alternative by guarantying information theoretical safety and relying on cryptographic assumptions only for liveness.

There exists other BAB protocols in the asynchronous setting. Some works like [29, 40] use cryptographic schemes for safety, while others like [19] do not use signatures. Other works like [22] encapsulate timing assumptions by relying on a failure detector. All these works have higher expected communication complexity.

The idea of building a communication DAG and locally interpreting total order was considered before [18, 21]. To the best of our knowledge, the only algorithms that realize this idea in the Byzantine settings are HashGraph [3] and later Aleph [24]. In contrast to DAG-Rider, HashGraph builds an unstructured DAG in which processes (unreliably) send messages with 2 references to previous vertices and on top of it run an inefficient binary agreement protocol, which leads to expected exponential time complexity. Their communication complexity is not straightforward to analyze since they did not clearly describe the mechanism that ensures that eventually all DAG information is propagated to all processes, and no analysis is provided. Aleph improves HashGraph's complexity by building a round-based DAG and using a more efficient binary agreement protocol [13] to agree on whether to commit every vertex in a round. They do not amortize complexity and have $O(n^3)$ per decision. In contrast to DAG-Rider, both HashGraph and Aleph (1) do not satisfy Validity; and (2) rely on signatures for safety and thus are not post-quantum safe.

## 8 CONCLUSION

We presented DAG-Rider: an asynchronous Byzantine Atomic Broadcast protocol with optimal resilience, optimal amortized communication complexity, and optimal time complexity. DAG-Rider does not rely on cryptographic assumptions for safety. Instead, it rules out Byzantine equivocation by relying on the reliable broadcast to guarantee that all correct processes eventually see the same DAG. Finally, we believe that DAG-Rider's elegant design, perfect load balancing, and modular separation of concerns make it an adequate candidate for future Byzantine SMR systems.

## ACKNOWLEDGMENTS

This work was funded by the Novi team at Facebook. We also wish to thank the Novi Research team for valuable feedback, and in particular George Danezis, Alberto Sonnino, and Dahlia Malkhi.

[2] It is also possible to store vertices hashes.

## REFERENCES

[1] 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* (Toronto ON, Canada) *(PODC '19)*. ACM, New York, NY, USA, 337–346. https://doi.org/10.1145/3293611.3331612

[2] 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*. ACM, 30.

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

[4] Laasya Bangalore, Ashish Choudhury, and Arpita Patra. 2018. Almost-surely terminating asynchronous Byzantine agreement revisited. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. 295–304.

[5] Shai Ben-David, Allan Borodin, Richard Karp, Gabor Tardos, and Avi Wigderson. 1994. On the power of randomization in on-line algorithms. *Algorithmica* 11, 1 (1994), 2–14.

[6] Michael Ben-Or and Ran El-Yaniv. 2003. Resilient-optimal interactive consistency in constant time. *Distributed Computing* 16, 4 (2003), 249–262.

[7] Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. 2020. Asynchronous Byzantine Agreement with Subquadratic Communication. In *Theory of Cryptography Conference*. Springer, 353–380.

[8] 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.

[9] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. 2009. Brahms: Byzantine resilient random membership sampling. *Computer Networks* 53, 13 (2009), 2340–2359.

[10] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. 2006. Randomized gossip algorithms. *IEEE/ACM Transactions on Networking (TON)* 14, SI (2006), 2508–2530.

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

[12] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*. Springer, 524–541.

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

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

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

[16] Ran Canetti and Tal Rabin. 1993. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 42–51.

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

[18] Gregory V Chockler, Nabil Huleihel, and Danny Dolev. 1998. An adaptive totally ordered multicast protocol that tolerates partitions. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*. 237–246.

[19] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2006. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *Comput. J.* 49, 1 (2006), 82–96.

[20] Danny Dolev and Eli Gafni. 2016. Some garbage in-some garbage out: Asynchronous t-Byzantine as asynchronous benign t-resilient system with fixed t-trojan-horse inputs. *arXiv preprint arXiv:1607.01210* (2016).

[21] Danny Dolev, Shlomo Kramer, and Dalia Malki. 1993. Early delivery totally ordered multicast in asynchronous environments. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE, 544–553.

[22] Assia Doudou, Benoit Garbinato, and Rachid Guerraoui. 2000. Abstractions for devising Byzantine-resilient state machine replication. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*. IEEE, 144–153.

[23] 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.

[24] Adam Gkagol, Damian Lesniak, Damian Straszak, and Michal Swiketek. 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.

[25] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2019. Scalable Byzantine Reliable Broadcast. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[26] Bruce M Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. 2010. Fast asynchronous Byzantine agreement and leader election with full information. *ACM Transactions on Algorithms (TALG)* 6, 4 (2010), 1–28.

[27] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. 2000. Randomized rumor spreading. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 565–574.

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

[29] Kim Potter Kihlstrom, Louise E Moser, and P Michael Melliar-Smith. 2001. The SecureRing group communication system. *ACM Transactions on Information and System Security (TISSEC)* 4, 4 (2001), 371–406.

[30] 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.

[31] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative Byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 45–58.

[32] Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. 2020. FairLedger: A Fair Blockchain Protocol for Financial Institutions. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik.

[33] 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.

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

[35] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. 2020. Dumbo-mvba: Optimal multi-valued validated asynchronous Byzantine agreement, revisited. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*. 129–138.

[36] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 31–42.

[37] Satoshi Nakamoto. 2008. *Bitcoin: A peer-to-peer electronic cash system*. Technical Report.

[38] Arpita Patra, Ashish Choudhury, and C Pandu Rangan. 2014. Asynchronous Byzantine agreement with optimal resilience. *Distributed computing* 27, 2 (2014), 111–146.

[39] Michael O Rabin. 1983. Randomized Byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. IEEE, 403–409.

[40] Michael K Reiter. 1994. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. 68–80.

[41] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

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

[43] Maofan Yin, Dahlia Malkhi, MK Reiter and, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *38th ACM symposium on Principles of Distributed Computing (PODC'19)*.