Embedding a Deterministic BFT Protocol in a Block DAG

Maria A Schett
mail@maria-a-schett.net
University College London
London, United Kingdom

George Danezis
g.danezis@ucl.ac.uk
University College London
London, United Kingdom

ABSTRACT

This work formalizes the structure and protocols underlying recent distributed systems leveraging block DAGs, which are essentially encoding Lamport's happened-before relations between blocks, as their core network primitives. We then present an embedding of any deterministic Byzantine fault tolerant protocol $\mathcal{P}$ to employ a block DAG for interpreting interactions between servers. Our main theorem proves that this embedding maintains all safety and liveness properties of $\mathcal{P}$. Technically, our theorem is based on the insight that a block DAG merely acts as an efficient reliable point-to-point channel between instances of $\mathcal{P}$ while also using $\mathcal{P}$ for efficient message compression.

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**.

KEYWORDS

blockchain, DAG, byzantine fault tolerant protocols

ACM Reference Format:
Maria A Schett and George Danezis. 2021. Embedding a Deterministic BFT Protocol in a Block 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, 10 pages. https://doi.org/10.1145/3465084.3467930

1 INTRODUCTION

Recent interest in blockchain and cryptocurrencies has resulted in a renewed interest in Byzantine fault tolerant consensus for state machine replication, as well as Byzantine consistent and reliable broadcast that is sufficient to build payment systems [2, 13]. These systems have high demands on throughput. To meet this demand a number of designs [23] depart from the traditional setting and generalize the idea of a blockchain to a more generic directed acyclic graph embodying Lamport's happened-before relations [18] between blocks. We refer to this structure as a *block DAG*. Instead of directly sending protocol messages to each other, participants rely on the common higher level abstraction of the block DAG. Moving from a chain to a graph structure allows for parallelization: every participant can propose a block with transactions and not merely a leader. As with blockchains, these blocks of transactions link cryptographically to past blocks establishing an order between them.

Examples of such designs are Hashgraph [1] used by the Hedera network, as well as Aleph [12], Blockmania [7], and Flare [21]. These works argue a number of advantages for the block DAG approach. First, maintaining a joint block DAG is simple and scalable, and can leverage widely-available distributed key-value stores. Second, they report impressive performance results compared with traditional protocols that materialize point-to-point messages as direct network messages. This results from batching many transactions in each block; using a low number of cryptographic signatures, having minimal overhead when running deterministic parts of the protocol; using a common block DAG logic while performing network IO, and only applying the higher-level protocol logic off-line possibly later; and as a result supporting running many instances of protocols in parallel 'for free'.

However, while the protocols may be simple and performant when implemented, their specification, and arguments for correctness, safety and liveness are far from simple. Their proofs and arguments are usually inherently tied to their specific applications and requirements, but both specification and formal arguments of Hashgraph, Aleph, Blockmania, and Flare are structured around two phases: *(i)* building a block DAG, and *(ii)* running a protocol on top of the block DAG. We generalize their arguments by giving an abstraction of a block DAG as a *reliable point-to-point link*. We can then rely on this abstraction to simulate a protocol $\mathcal{P}$—as a black-box—on top of this point-to-point link *maintaining the safety and liveness properties* of $\mathcal{P}$. We believe that this modular formulation of the underlying mechanisms through a clear separation of the high-level protocol $\mathcal{P}$ and the underlying block DAG allows for easy re-usability and strengthens the foundations and persuasiveness of systems based on block DAGs.

In this work we present a formalization of a block DAG, the protocols to maintain a joint block DAG, and its properties. We show that any deterministic Byzantine fault tolerant (BFT) protocol, can be embedded in this block DAG, while maintaining its safety and liveness properties. We demonstrate that the advantageous properties of block DAG based protocols claimed by Hashgraph, Aleph, Blockmania, and Flare—such as the efficient message compression, batching of signatures, the ability to run multiple instances 'for free', and off-line interpretation of the block DAG—emerge from the generic composition we present. Therefore, the proposed composition not only allows for straight forward correctness arguments, but also preserves the claimed advantages of using a block DAG approach, making it a useful abstraction not only to analyze but also implement systems that offer both high assurance and high performance.

PODC '21, July 26–30, 2021, Virtual Event, Italy

ACM ISBN 978-1-4503-8548-0/21/07…$15.00
https://doi.org/10.1145/3465084.3467930

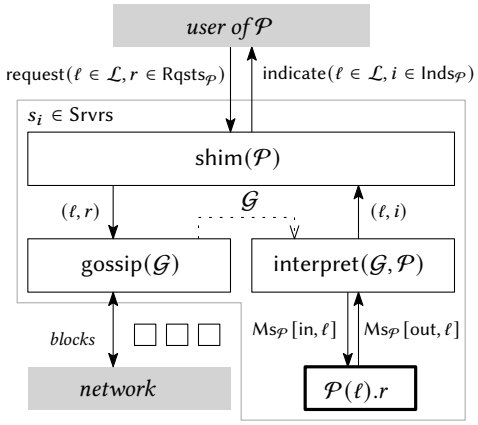

Figure 1: Components and interfaces.

Overview. Figure 1 shows the interfaces and components of our proposed block DAG framework parametric by a deterministic BFT protocol $\mathcal{P}$. At the top, we have a *user* seeking to run one or multiple instances of $\mathcal{P}$ on servers Srvrs. First, to distinguish between multiple *protocol instances* the user assigns them a *label* ℓ from a set of *labels* $\mathcal{L}$. Now, for $\mathcal{P}$ there is a set of possible *requests* $\text{Rqsts}_{\mathcal{P}}$. But instead of requesting $r \in \text{Rqsts}_{\mathcal{P}}$ from $s_i \in$ Srvrs running $\mathcal{P}$ for protocol instance ℓ, the user calls the high-level interface of our block DAG framework: request(ℓ, r) in shim($\mathcal{P}$). Internally, s_i passes (ℓ, r) on to gossip($\mathcal{G}$)—which continuously builds s_i's *block DAG* $\mathcal{G}$ by receiving and disseminating *blocks*. The passed (ℓ, r) is included into the next block s_i disseminates, and s_i also includes references to other received blocks, where cryptographic primitives prevent byzantine servers from adding cycles between blocks [19]. These blocks are continuously exchanged by the servers utilizing the low-level interface to the *network* to exchange blocks. In Section 3 we formally define the block DAG, its properties and protocols for servers to maintain a joint block DAG. Independently, indicated by the dotted line, s_i interprets $\mathcal{P}$ by *reading* $\mathcal{G}$ and running interpret$(\mathcal{G}, \mathcal{P})$. To do so, s_i locally simulates every protocol instance $\mathcal{P}$ with label ℓ by simulating one process instance of $\mathcal{P}(\ell)$ for every server $s \in$ Srvrs. To drive the simulation, s_i passes the request r read from a block in $\mathcal{G}$ to $\mathcal{P}$, and then s_i simulates the message exchange between any two servers based on the structure of the block DAG and the deterministic protocol $\mathcal{P}$. Therefore s_i moves messages between in- and out-buffers $\text{Ms}_{\mathcal{P}}[\text{in}, \ell]$ and $\text{Ms}_{\mathcal{P}}[\text{out}, \ell]$. Eventually, the simulation $\mathcal{P}(\ell)$ of the server s_i will indicate i from the set of possible *indications* $\text{Inds}_{\mathcal{P}}$. We show how the block DAG essentially acts as a reliable point-to-point link and describe how any deterministic BFT protocol $\mathcal{P}$ can be interpreted on a block DAG in Section 4. Finally, after interpret indicated i, shim($\mathcal{P}$) can indicate i for ℓ to the user of $\mathcal{P}$. From the user's perspective, the embedding of $\mathcal{P}$ acted as $\mathcal{P}$, *i.e.* shim($\mathcal{P}$) *maintained* $\mathcal{P}$*'s interfaces and properties*. We prove this in Section 5 and illustrate the block DAG framework for $\mathcal{P}$ instantiated with byzantine reliable broadcast protocol. We give related work in Section 6, and conclude in Section 7, where we discuss integration aspects of higher-level protocols and the block DAG framework—including challenges in embedding protocols with non-determinism, more advanced cryptography, and BFT protocols operating under partial synchrony.

This is the short version of the paper, where we omit proof details and the appendix. Please find the full version on ARXIV [22].

Contributions. We show that using the block DAG framework of Figure 1 for a deterministic BFT protocol $\mathcal{P}$ maintains the *(i)* interfaces, and *(ii)* safety and liveness properties of $\mathcal{P}$ (Theorem 5.1). The argument is generic: interpreting the eventually *joint block DAG* implements a *reliable point-to-point link* (Lemma 3.7, Lemma 4.3). Using this reliable point-to-point link any server can locally run a simulation of $\mathcal{P}$ as a black-box. This simulation is an execution of $\mathcal{P}$ and thus retains the properties of $\mathcal{P}$. By using the block DAG framework, the user gains efficient message compression and runs instances of $\mathcal{P}$ in parallel 'for free'. This is due to the determinism of $\mathcal{P}$, which allows every server to locally deduce message contents from only the fact that a message has been received without actually receiving the contents. So every received block automatically compresses these messages for preceding blocks. Moreover, with every new block every server creates a new instance of $\mathcal{P}$.

2 BACKGROUND

System Model. We assume a finite set of *servers* Srvrs. A *correct* server $s \in$ Srvrs faithfully follows a protocol $\mathcal{P}$. When s is *byzantine*, then s behaves arbitrarily. However, we assume byzantine servers are computationally bounded (*e.g.*, s cannot forge signatures, or find collisions in cryptographic hash functions) and cannot interfere with the Trusted Computing Base of correct servers (*e.g.*, kill the electricity supply of correct servers). The set Srvrs is fixed and known by every $s' \in$ Srvrs and we assume $3f + 1$ servers to tolerate at most f byzantine servers. The set of all messages in protocol $\mathcal{P}$ is $\mathsf{M}_{\mathcal{P}}$. Every message $m \in \mathsf{M}_{\mathcal{P}}$ has a m.sender and a m.receiver. We assume an arbitrary, but fixed, total order on messages: $<_{\mathsf{M}}$. A protocol $\mathcal{P}$ is *deterministic* if a state q and a sequence of messages $m \in \mathsf{M}_{\mathcal{P}}$ determine state q' and out-going messages $M \subseteq 2^{\mathsf{M}_{\mathcal{P}}}$. In particular, deterministic protocols do not rely on random behavior such as coin-flips. The exact requirements on the network synchronicity depend on the protocol $\mathcal{P}$, that we want to embed, *e.g.*, we may require partial synchrony [9] to avoid FLP [10]. The only network assumption we impose for building block DAGs is the following:

ASSUMPTION 1 (RELIABLE DELIVERY). *For two correct servers s_1 and s_2, if s_1 sends a block B to s_2, then eventually s_2 receives B.*

Cryptographic Primitives. We assume a *secure cryptographic hash function* $\# : A \to A'$ and write $\#(x)$ for the hash of $x \in A$, and $\#(A)$ for A' [22, Definition A.1]. We further assume a secure *cryptographic signature* scheme [15]: given a set of *signatures* Σ we have functions sign : Srvrs $\times \mathsf{M} \to \Sigma$ and verify : Srvrs $\times \mathsf{M} \times \Sigma \to \mathbb{B}$, where verify$(s, m, \sigma) =$ true iff sign$(s, m) = \sigma$. Given computational bounds on all participants appropriate parameters for these schemes can be chosen to make their probability of failure negligible, and for the remainder of this work we assume their probability of failure to be zero.

Directed Acyclic Graphs. A *directed graph* $\mathcal{G}$ is a pair of *vertices* V and *edges* $\mathsf{E} \subseteq \mathsf{V} \times \mathsf{V}$. We write $\varnothing$ for the *empty graph*. If there

is an edge from v to v', that is $(v, v') \in \mathsf{E}$, we write $v \rightharpoonup v'$. If v' is *reachable* from v, then (v, v') is in the transitive closure of $\rightharpoonup$, and we write $\rightharpoonup^+$. We write $\rightharpoonup^*$ for the reflexive and transitive closure, and $v \rightharpoonup^n v'$ for $n \geqslant 0$ if v' is reachable from v in n steps. A graph $\mathcal{G}$ is *acyclic*, if $v \rightharpoonup^+ v'$ implies $v \neq v'$ for all nodes $v, v' \in \mathcal{G}$. We abbreviate $v \in \mathcal{G}$ if $v \in \mathsf{V}_\mathcal{G}$, and $V \subseteq \mathcal{G}$ if $v \in \mathcal{G}$ for all $v \in V$. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be directed graphs. We define $\mathcal{G}_1 \cup \mathcal{G}_2$ as $(\mathsf{V}_{\mathcal{G}_1} \cup \mathsf{V}_{\mathcal{G}_2}, \mathsf{E}_{\mathcal{G}_1} \cup \mathsf{E}_{\mathcal{G}_2})$, and $\mathcal{G}_1 \leqslant \mathcal{G}_2$ holds if $\mathsf{V}_{\mathcal{G}_1} \subseteq \mathsf{V}_{\mathcal{G}_2}$ and $\mathsf{E}_{\mathcal{G}_1} = \mathsf{E}_{\mathcal{G}_2} \cap (\mathsf{V}_{\mathcal{G}_1} \times \mathsf{V}_{\mathcal{G}_1})$. Note, for $\leqslant$ we not only require $\mathsf{E}_{\mathcal{G}_1} \subseteq \mathsf{E}_{\mathcal{G}_2}$, but additionally $\mathsf{E}_{\mathcal{G}_1}$ must already contain all edges from $\mathsf{E}_{\mathcal{G}_2}$ between vertices in $\mathcal{G}_1$. The following definition for inserting a new vertex v is restrictive: it permits to extend $\mathcal{G}$ only by a vertex v and edges to this v.

Definition 2.1. Let $\mathcal{G}$ be a directed graph, v be a vertex, and a E be a set of edges of the form $\{(v_i, v) \mid v_i \in V \subseteq \mathcal{G}\}$. We define $\mathsf{insert}(\mathcal{G}, v, E) = (\mathsf{V}_\mathcal{G} \cup \{v\}, \mathsf{E}_\mathcal{G} \cup E)$.

This unconventional definition of inserting a vertex is sufficient for building a block DAG—and helps to establish useful properties of the block DAG in the next lemma: (1) inserting a vertex is idempotent, (2) the original graph is a subgraph of the graph with a newly inserted vertex, and (3) a block DAG is acyclic by construction.

Lemma 2.2. *For a directed graph $\mathcal{G}$, a vertex v, and a set of edges $E = \{(v_i, v) \mid v_i \in V \subseteq \mathcal{G}\}$, the following properties of* $\mathsf{insert}(\mathcal{G}, v, E)$ *hold: (1) if $v \in \mathcal{G}$ and $E \subseteq \mathsf{E}_\mathcal{G}$, then* $\mathsf{insert}(\mathcal{G}, v, E) = \mathcal{G}$*; (2) if $E = \{(v_i, v) \mid v_i \in V \subseteq \mathcal{G}\}$ and $v \notin \mathcal{G}$, then $\mathcal{G} \leqslant$* $\mathsf{insert}(\mathcal{G}, v, E)$*; and (3) if $\mathcal{G}$ is acyclic, $v \notin \mathcal{G}$, then* $\mathsf{insert}(\mathcal{G}, v, E)$ *is acyclic.*

To give some intuitions, for Lemma 2.2 (2), if $v \in \mathcal{G}$ and $\mathcal{G}' = \mathsf{insert}(\mathcal{G}, v, E)$, then $\mathsf{E}_{\mathcal{G}'} \cap (\mathsf{V}_\mathcal{G} \times \mathsf{V}_\mathcal{G}) = \mathsf{E}_\mathcal{G}$ may not hold. For example, let $\mathcal{G}$ have vertices v_1 and v_2 with $\mathsf{E}_\mathcal{G} = \emptyset$, and $\mathcal{G}' = \mathsf{insert}(\mathcal{G}, v_2, \{(v_1, v_2)\})$ with $\mathsf{E}_{\mathcal{G}'} = \{(v_1, v_2)\}$. Then we have $\mathsf{E}_\mathcal{G} \neq \mathsf{E}_{\mathcal{G}'} \cap (\mathsf{V}_\mathcal{G} \times \mathsf{V}_\mathcal{G})$. For Lemma 2.2 (3), if $v \in \mathcal{G}$, then $\mathsf{insert}(\mathcal{G}, v, E)$ may add a cycle. For example, take $\mathcal{G}$ with vertices $\{v_1, v_2\}$ and $\mathsf{E}_\mathcal{G} = \{(v_1, v_2)\}$ then $\mathsf{insert}(\mathcal{G}, v_1, \{(v_2, v_1)\})$ contains a cycle.

3 BUILDING A BLOCK DAG

The networking component of the block DAG protocol is very simple: it has one core message type, namely a *block*, which is constantly disseminated. A block contains authentication for references to previous blocks, requests associated to instances of protocol $\mathcal{P}$, meta-data and a signature. Servers only exchange and validate blocks. From these blocks with their references to previous blocks, servers build their block DAGs. Although servers build their block DAGs locally, eventually correct servers have a joint block DAG $\mathcal{G}$. As we show in the next Section 4, the servers can then *independently* interpret $\mathcal{G}$ as multiple instances of $\mathcal{P}$.

Definition 3.1. A *block* $B \in \mathsf{Blks}$ has *(i)* an *identifier* n of the server s which built B, *(ii)* a *sequence number* $\mathsf{k} \in \mathbb{N}_0$, *(iii)* a finite list of hashes of *predecessor* blocks $\mathsf{preds} = [\mathsf{ref}(B_1), \ldots, \mathsf{ref}(B_k)]$, *(iv)* a finite list of labels and requests $\mathsf{rs} \in 2^{\mathcal{L} \times \mathsf{Rqsts}}$, and *(v)* a *signature* $\sigma = \mathsf{sign}(\mathsf{n}, \mathsf{ref}(B))$. Here, ref is a secure cryptographic hash function computed from n, k, preds, and rs, but not σ. By not depending on σ, $\mathsf{sign}(B.\mathsf{n}, \mathsf{ref}(B))$ is well defined.

We use B and $\mathsf{ref}(B)$ interchangeably, which is justified by *collision resistance* of ref [22, Definition A.1(3)]. We use register notation, *e.g.*, $B.\mathsf{n}$ or $B.\sigma$, to refer to elements of a block B, and abbreviate $B' \in \{B' \mid \mathsf{ref}(B') \in B.\mathsf{preds}\}$ with $B' \in B.\mathsf{preds}$. Given blocks B and B' with $B.\mathsf{n} = B'.\mathsf{n}$ and $B'.\mathsf{k} = B.\mathsf{k} + 1$. If $B \in B'.\mathsf{preds}$ then we call B a *parent* of B' and write $B'.\mathsf{parent} = B$. We require that every block has at most one parent. We call B a *genesis block* if $B.\mathsf{k} = 0$. A genesis block B cannot have a parent block, because $B.\mathsf{k} = 0$ and 0 is minimal in $\mathbb{N}_0$.

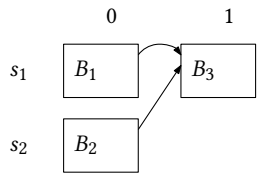

Figure 2: A block DAG with 3 blocks B_1, B_2, and B_3.

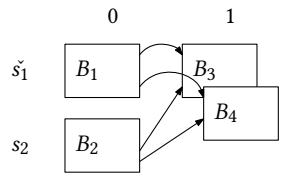

Figure 3: A block DAG, where $\check{s}_1$ is equivocating.

Lemma 3.2. *For blocks B_1 and B_2, if $B_1 \in B_2.\mathsf{preds}$ then $B_2 \notin B_1.\mathsf{preds}$.*

Lemma 3.2 prevents a byzantine server $\check{s}$ to include a cyclic reference between $\check{B}$ and B by (1) waiting for—or building itself—a block B with $\mathsf{ref}(\check{B}) \in B.\mathsf{preds}$, and then (2) building a block $\check{B}$ such that $\mathsf{ref}(\check{B}) \in B$. As with secure time-lines [19], Lemma 3.2 gives a temporal ordering on B and $\check{B}$. This is a static, cryptographic property, based on the security of hash functions, and not dependent on the order in which blocks are received on a network. While this prevents byzantine servers from introducing cycles, they can still build "faulty" blocks. So next we define three checks for a server to ascertain that a block is well-formed. If a block passes these checks, the block is *valid* from this server's point of view and the server *validated* the block.

Definition 3.3. A server s considers a block B *valid*, written $\mathsf{valid}(s, B)$, if *(i)* s confirms $\mathsf{verify}(B.\mathsf{n}, B.\sigma)$, *i.e.*, that $B.\mathsf{n}$ built B, *(ii)* either *(a)* B is a genesis block, or *(b)* B has exactly one parent, and *(iii)* s considers all blocks $B' \in B.\mathsf{preds}$ valid.

Especially, *(ii)* deserves our attention: a server $\check{s}_1$ may still build two different blocks having the same parent. However, $\check{s}_1$ will not be able to create a further block to 'join' these two blocks with a different parent—their successors will remain split. Essentially, this forces a linear history from every block.

We assume, that if a correct server s considers a block B valid, then s can forward any block $B' \in B.\mathsf{preds}$. That is, s has received the full content of B'—not only $\mathsf{ref}(B')$—and persistently stores B'. From valid blocks and their predecessors, a correct server builds a *block DAG*:

Definition 3.4. For a server s, a *block DAG* $\mathcal{G} \in \mathsf{Dags}$ is a directed acyclic graph with vertices $\mathsf{V}_\mathcal{G} \subseteq \mathsf{Blks}$, where *(i)* $\mathsf{valid}(s, B)$ holds for all $B \in \mathsf{V}_\mathcal{G}$, and *(ii)* if $B \in B'.\mathsf{preds}$ then $B \in \mathsf{V}_\mathcal{G}$ and $(B, B') \in \mathsf{E}_\mathcal{G}$ holds for all $B' \in \mathsf{V}_\mathcal{G}$. Let B' be a block such that $\mathsf{valid}(s, B')$ holds and $B \in \mathcal{G}$ for all $B \in B'.\mathsf{preds}$. Then s *inserts* B' into $\mathcal{G}$ by $\mathsf{insert}(\mathcal{G}, B', \{(B, B') \mid B \in B'.\mathsf{preds}\})$ after Definition 2.1 and we write $\mathcal{G}.\mathsf{insert}(B)$. The preconditions guarantee that $\mathcal{G}.\mathsf{insert}(B')$ is a block DAG [22, Lemma A.3].

Example 3.5. In Figure 2 we show a block DAG with three blocks B_1, B_2, and B_3, where $B_1 = \{\mathsf{n} = s_1, \mathsf{k} = 0, \mathsf{preds} = [\,]\}$, $B_2 =$

```
1  module gossip(s ∈ Srvrs, 𝒢 ∈ Dags, rqsts ∈ 2^(ℒ×Rqsts))
2      ℬ := {n : s, k : 0, preds : [ ], rs : [ ], σ : null} ∈ Blks
3      blks := ∅ ∈ 2^Blks
4      when received B ∈ Blks and B ∉ 𝒢
5          blks := blks ∪ {B}
6      when valid(s, B′) for some B′ ∈ blks
7          𝒢.insert(B′)
8          ℬ.preds := ℬ.preds · [ref(B′)]
9          blks := blks \ {B′}
10     when B′ ∈ blks and B ∈ B′.preds where B ∉ blks and
         B ∉ 𝒢
11         send FWD ref(B) to B′.n
12     when received FWD ref(B) from s′ and B ∈ 𝒢
13         send B to s′
14     when disseminate()
15         ℬ := {ℬ with rs : rqsts.get(), σ : sign(s, ℬ)}
16         𝒢.insert(ℬ)
17         send ℬ to every s′ ∈ Srvrs
18         ℬ := {n : s, k : ℬ.k + 1, preds : [ref(ℬ)], rs : [ ],
             σ : null}
```

Algorithm 1: Building the block DAG $\mathcal{G}$ and block $\mathcal{B}$.

$\{n = s_2, k = 0, preds = [\,]\}$, and $B_3 = \{n = s_1, k = 1, preds = [ref(B_1), ref(B_2)]\}$. Here, $parent(B_3) = B_1$. Consider now Figure 3 adding the block: $B_4 = \{n = s_1, k = 1, preds = [ref(B_1), ref(B_2)]\}$. While all blocks in Figure 3 are valid, with block B_4, s_1' is *equivocating* on the block B_3—and vice versa.

To build a block DAG and blocks every correct server follows the gossip protocol in Algorithm 1. By building a block DAG every correct server will eventually have a joint view on the system. By building a block, every server can inject messages into the system: either explicit messages from the high-level protocol by directly writing those into the block, or implicit messages by adding references to other blocks. In Algorithm 1, a server s builds *(i)* its block DAG $\mathcal{G}$ in lines 4–13, and *(ii)* its current block $\mathcal{B}$ by including requests and references to other blocks in lines 14–18. The servers communicate by exchanging blocks. Assumption 1 guarantees, that a correct s will eventually receive a block from another correct server. Moreover, every correct server s will regularly request disseminate() in line 14 and will eventually send their own block $\mathcal{B}$ in line 17. This is guaranteed by the high-level protocol (*cf.* Section 5).

Every server s operates on four data structures. The two data structures which are shared with Algorithm 2 are given as arguments in line 1: *(i)* the block DAG $\mathcal{G}$, which Algorithm 2 will only read, and *(ii)* a buffer rqsts, where Algorithm 2 inserts pairs of labels and requests. On the other hand, s also keeps *(iii)* the block $\mathcal{B}$ which s currently builds (line 2), and *(iv)* a buffer blks of received blocks (line 3). To build its block DAG, s inserts blocks into $\mathcal{G}$ in line 7 and line 16. It is guaranteed that by inserting those blocks $\mathcal{G}$ remains a block DAG [22, Lemma A.5]. To insert a block, s keeps track of its received blocks as candidate blocks in the buffer blks (line 4–5). Whenever s considers a $B' \in$ blks valid (line 6), s inserts B' in $\mathcal{G}$ (line 7). However, to consider a block B' valid, s has to consider all its predecessors valid—and s may not have yet received every $B \in B'$.preds. That is, $B' \in$ blks but $B \notin$ blks and $B \notin \mathcal{G}$ (*cmp.* line 10). Now, s can request forwarding of B from the server that built B', *i.e.* from s' where $B'.n = s'$, by sending FWD B to s' (lines 10–11). To prevent s from flooding s' an implementation would guard lines 10–11, *e.g.* by a timer $\Delta_{B'}$. That is, we implicitly assume that for every block B' a correct server waits a reasonable amount of time before (re-)issuing a forward request. The wait time should be informed by the estimated round-trip time and can be adapted for repeating forwarding requests.

On the other hand, s also answers to forwarding requests for a block B from s', where $B \in B'$.preds of some block B' disseminated by s (lines 12–13). It is not necessary to request forwarding from servers other than s'. We only require that correct servers will eventually share the same blocks. This mechanism, together with Assumption 1 and s's eventual dissemination of $\mathcal{B}$, allows us to establish the following lemma:

Lemma 3.6. *For a correct server s executing* gossip*, if s receives a block B, which s considers valid, then (1) every correct server will eventually receive B, and (2) every correct server will eventually consider B valid.*

In parallel to building $\mathcal{G}$, s builds its current block $\mathcal{B}$ by *(i)* continuously adding a reference to any block B', which s receives and considers valid in line 8 (adding at most one reference to B' [22, Lemma A.6]), and *(ii)* eventually sending $\mathcal{B}$ to every server in line 17. Just before s sends $\mathcal{B}$, s injects literal inscriptions of $(\ell_i, r_i) \in$ rqsts into $\mathcal{B}$ in line 15. Now rs holds requests r_i for the protocol instances $\mathcal{P}$ with label ℓ_i. These requests will eventually be read in Algorithm 2. Finally, s signs $\mathcal{B}$ in line 15, sends $\mathcal{B}$ to every server, and starts building its next $\mathcal{B}$ in line 18 by incrementing the sequence number k, initializing preds with the parent block, as well as clearing rs and σ.

So far we established, how s builds its own block DAG. Next we want to establish the concept of a *joint block DAG* between two correct servers s and s'. Let $\mathcal{G}_s$ and $\mathcal{G}_{s'}$ be the block DAG of s and s'. We define their *joint block DAG* $\mathcal{G}'$ as a block DAG $\mathcal{G}' \geqslant \mathcal{G}_s \cup \mathcal{G}_{s'}$. This joint block DAG is a block DAG for s and for s' [22, Lemma A.7]. Intuitively, we want any two correct servers to be able to 'gossip some more' and arrive at their joint block DAG $\mathcal{G}'$.

Lemma 3.7. *Let s and s' be correct servers with block DAGs $\mathcal{G}_s$ and $\mathcal{G}_{s'}$. By executing* gossip *in Algorithm 1, eventually s has a block DAG $\mathcal{G}_s'$ such that $\mathcal{G}_s' \geqslant \mathcal{G}_s \cup \mathcal{G}_{s'}$.*

Proof. By [22, Lemma A.5] any block DAG $\mathcal{G}'$ obtained through gossip is a block DAG, and by [22, Lemma A.7] $\mathcal{G}'$ is a block DAG for s. It remains to show that by executing gossip, eventually $\mathcal{G}'$ will be the block DAG for s. As s' received and considers all $B \in \mathcal{G}_{s'}$ valid, by Lemma 3.6 (2) s will eventually consider every B valid. By executing gossip, s will eventually insert every B in its block DAG and $\mathcal{G}'$ will contain all $B \in \mathcal{G}_{s'}$. □

In the next section, we will show how s and s' can independently interpret a deterministic protocol $\mathcal{P}$ on this joint block DAG. But before we do so, we want to highlight that the gossip protocol retains the key benefits reported by works using the block DAG approach,

namely simplicity and amenability to high-performance implementation. Currently, our gossip protocol in Algorithm 1 uses an explicit forwarding mechanism in lines 10–13. This explicit forwarding mechanism—as opposed to every correct server re-transmitting *every* received and valid block in a second communication round—is possible through blocks including references to predecessor blocks. Hence, every server knows what blocks it is missing and whom to ask for them. But in an implementation, we would go a step further and replace the forwarding mechanism—and messages—as described next: each block is associated with a unique cryptographic reference that authenticates its content. As a result both best-effort broadcast operations as well as synchronization operations can be implemented using distributed and scalable key-value stores at each server (*e.g.*, Appache Cassandra, Aws S3), which through sharding have no limits on their throughput. Best-effort broadcasts can be implemented directly, through simple asynchronous IO. This is due to the the (now) single type of message, namely blocks, and a single handler for blocks in gossip that performs minimal work: it just records blocks, and then asynchronously ensures their predecessors exist (potentially doing a remote key-value read) and they are valid (which only involves reference lookups into a hash-table and a single signature verification). Alternatively, best-effort broadcast itself can be implemented using a publish-subscribe notification system and remote reads into distributed key value stores. In summary, the simplicity and regularity of gossip, and the weak assumptions and light processing allow systems' engineers great freedom to optimize and distribute a server's operations. Both Hashgraph and Blockmania (which have seen commercial implementation) report throughputs of many 100,000 transactions per second, and latency in the order of seconds. As we will see in the next section no matter which $\mathcal{P}$ the servers s and s' choose to interpret, they can build a joint block DAG using the same gossip logic—by only exchanging blocks—and then *independently* interpreting $\mathcal{P}$ on $\mathcal{G}$.

4 INTERPRETING A PROTOCOL

Every server s interprets the protocol $\mathcal{P}$ embedded in its local block DAG $\mathcal{G}$. This interpretation is completely *decoupled* from building the block DAG in Algorithm 1. To interpret one *protocol instance* of $\mathcal{P}$ tagged with label ℓ, server s locally runs one *process instance* of $\mathcal{P}$ with label ℓ for every other server $s_i \in$ Srvrs. Thereby, s treats $\mathcal{P}$ as a black-box which *(i)* takes a request or a message, and *(ii)* returns messages or an indication. A server s can fully simulate the protocol instance $\mathcal{P}$ for any other server because their requests and messages have been embedded in the block DAG $\mathcal{G}$ by Algorithm 1. User requests r_j to $\mathcal{P}$ are embedded in a block $B \in \mathcal{G}$ in B.rs and s reads these requests from the block and passes them on to the simulation of $\mathcal{P}$. Since $\mathcal{P}$ is deterministic, s can—after the initial request r_j for $\mathcal{P}$—compute all subsequent messages which would have been sent in $\mathcal{P}$ by interpreting edges between blocks, such as $B_1 \rightharpoonup B_2$, as messages sent from B_1.n to B_2.n. There is no need for explicitly sending these messages. And indeed, our goal is to show that the interpretation of a deterministic protocol $\mathcal{P}$ embedded in a block DAG implements a reliable point-to-point link.

To treat $\mathcal{P}$ as a black-box, we assume the following high-level interface: *(i)* an interface to *request* $r \in \mathsf{Rqsts}_{\mathcal{P}}$, and *(ii)* an interface where $\mathcal{P}$ *indicates* $i \in \mathsf{Inds}_{\mathcal{P}}$. When a request r reaches a process instance, we assume that it immediately returns messages $m_1, \ldots, m_k$ triggered by r. This is justified, as s runs all process instances locally. As requests do not depend on the state of the process instance, also these messages do not depend on the current state of process instance. We also assume a low-level interface for $\mathcal{P}$ to *receive* a message m. Again, we assume that when m reaches a process instance, it immediately returns the messages $m_1, \ldots, m_k$ triggered by m.

```
1  module interpret(G ∈ Dags, P ∈ module)
2      I[B ∈ Blks] := false ∈ 𝔹
3      when B ∈ G where eligible(B)
4          B.PIs := copy B.parent.PIs
5          for every (ℓ_j ∈ ℒ, r_j ∈ Rqsts) ∈ B.rs
6              B.Ms[out, ℓ_j] := B.PIs[ℓ_j].r_j
7          for every
             ℓ_j ∈ {ℓ_j | (ℓ_j, r_j) ∈ B_j.rs ∧ B_j ∈ G ∧ B_j ⇀⁺ B}
8              for every B_i ∈ B.preds
9                  B.Ms[in, ℓ_j] := B.Ms[in, ℓ_j] ∪ {m |
                     B_i.Ms[out, ℓ_j] and m.receiver = B.n}
10             for every m ∈ B.Ms[in, ℓ_j] ordered by <_M
11                 B.Ms[out, ℓ_j] :=
                     B.Ms[out, ℓ_j] ∪ B.PIs[ℓ_j].receive(m)
12         I[B] = true
13     when B.PIs[ℓ_j].i
14         indicate(ℓ_j, i, B.n)
```

Algorithm 2: Interpreting protocol $\mathcal{P}$ on the block DAG $\mathcal{G}$.

Algorithm 2 shows the protocol executed by s for interpreting a deterministic protocol $\mathcal{P}$ on a block DAG $\mathcal{G}$. The key task is to 'get messages from one block and give them to the next block'. Therefore s traverses through every $B \in \mathcal{G}$. To keep track of which blocks in $\mathcal{G}$ it has already interpreted, s uses $\mathcal{I}$ in line 2. Note, that edges in $\mathcal{G}$ impose a partial order: s considers a block $B \in \mathcal{G}$ as eligible(B) for interpretation if *(i)* $\mathcal{I}[B] =$ false, and *(ii)* for every $B_i \in B$.preds, $\mathcal{I}[B_i] =$ true holds. While there may be more than one B eligible, every $B \in \mathcal{G}$ is interpreted eventually [22, Lemma A.10]. Now s picks an eligible B in line 3 and *interprets* B in lines 4–12. To interpret B, s needs to keeps track of two variables for every protocol instance ℓ_j: (1) the state of the process instance ℓ_j for a server $s_i \in$ Srvrs in PIs$[\ell_j]$, and (2) the state of in-going and out-going messages in Ms$[\mathsf{in}, \ell_j]$ and Ms$[\mathsf{out}, \ell_j]$.

Our goal is to track changes to these two variables—the process instances PIs and message buffers Ms—throughout the interpretation of $\mathcal{G}$. To do so, we assign their state to every block B. Before B is interpreted, we assume B.PIs$[\ell_j]$ to be initialized with $\bot$, and B.Ms$[d \in [\{\mathsf{in}, \mathsf{out}\}, \ell_j]$ with $\varnothing$. They remain so while B is eligible [22, Lemma A.15].

After interpreting B, (1) B.PIs$[\ell_j]$ holds the state of the process instance ℓ_j of the server s_i, which built B, *i.e.*, $s_i = B$.n, and

(2) $B.\mathsf{Ms}[\mathsf{in}, \ell_j]$ holds the in-going messages for s_i and $\mathsf{Ms}[\mathsf{out}, \ell_j]$ the out-going messages from s_i for process instance ℓ_j[1].

As a starting point for computing the state of $B.\mathsf{PIs}[\ell_j]$, s copies the state from the parent block of B in line 4. For the base case, *i.e.* all (genesis) blocks B without parents, we assume $B.\mathsf{PIs}[\ell_j] :=$ **new process** $\mathcal{P}(\ell_j, s_i)$ where $s_i = B.\mathsf{n}$. This is effectively a simplification: we assume a running process instance ℓ_j for every $s_i \in \mathsf{Srvrs}$. In an implementation, we would only start process instances for ℓ_j after receiving the first message or request for ℓ_j for $s_i = B.\mathsf{n}$. Now in our simplification, we start all process instances for every label at the genesis blocks and pass them on from the parent blocks. This leads us to our step case: B has a parent. As $B.\mathsf{parent} \in B.\mathsf{preds}$, $B.\mathsf{parent}$ has been interpreted and moreover $B.\mathsf{parent.n} = s_i$ [22, Lemma A.13]. Next, to advance the copied state on B, s processes (1) all incoming requests r_j given by $B.\mathsf{rs}$ in lines 5–6, and (2) all incoming messages from $B_i.\mathsf{n}$ to $B.\mathsf{n}$ given by $B_i \rightharpoonup B$ in lines 8–11. For the former (1), s reads the labels and requests from the field $B.\mathsf{rs}$. Here r_j is the literal transcription of the user's original request given to $\mathcal{P}$. To give an example, if $\mathcal{P}$ is reliable broadcast, then r_j could read 'broadcast(42)' (*cf.* Section 5). When interpreting, s requests r_j from $B.\mathsf{n}$'s simulated protocol instance: $B.\mathsf{PIs}[\ell_j].r_j$. For the latter (2), s collects *(i)* in $B.\mathsf{Ms}[\mathsf{in}, \ell]$ all messages for $B.\mathsf{n}$ from $B_i.\mathsf{Ms}[\mathsf{out}, \ell]$ where $B_i \in B.\mathsf{preds}$ in lines 8–9 and then feeds *(ii)* $m \in B.\mathsf{Ms}[\mathsf{in}, \ell]$ to $B.\mathsf{PIs}[\ell]$ in lines 10–11 in order $<_{\mathsf{M}}$. This (arbitrary) order is a simple way to guarantee that every server interpreting Algorithm 2 will execute exactly the same steps. By feeding those messages and requests to $B.\mathsf{PIs}[\ell_j]$ in lines 6 and 11 s computes (1) the next state of $B.\mathsf{PIs}[\ell_j]$ and (2) the out-going messages from $B.\mathsf{n}$ in $B.\mathsf{Ms}[\mathsf{out}, \ell_j]$. By construction, $m.\mathsf{sender} = B.\mathsf{n}$ for $m \in B.\mathsf{Ms}[\mathsf{out}, \ell_j]$ [22, Lemma A.14]. Once, s has completed this, s marks B as interpreted in line 12 and can move on to the next eligible block. After s interpreted B, the simulated process instance $B.\mathsf{PIs}[\ell_j]$ may indicate $i \in \mathsf{Inds}$. If this is the case, s indicates i for ℓ_j on behalf of $B.\mathsf{n}$ in lines 13–14. Note, that none of the steps used the fact that it was s who interpreted $B \in \mathcal{G}$. So, for every B, every $s' \in \mathsf{Srvrs}$ will come to the exact same conclusion.

But we glossed over a detail, s actually had to take a choice—more than one B may have been eligible in line 3. This is a feature: by having this choice we can think of interpreting a $\mathcal{G}'$ with $\mathcal{G}' \geqslant \mathcal{G}$ as an 'extension' of interpreting $\mathcal{G}$. And, for two eligible B_1 and B_2 it does not matter if we pick B_1 before B_2. Informally, this is because when we pick B_1 in line 3, only the the state with respect to B_1 is modified—and this state does not depend on B_2 [22, Lemma A.11]. Another detail we glossed over is line 7: when interpreting B, s interprets the process instances of every ℓ_j relevant on B *at the same time*. But again, because $\ell_j \neq \ell'_j$ are independent instances of the protocol with disjoint messages, *i.e.*, $B_i.\mathsf{Ms}[\mathsf{out}, \ell_j]$ in line 9 is independent of any $B_i.\mathsf{Ms}[\mathsf{out}, \ell'_j]$, they do not influence each other and the order in which we process ℓ_j does not matter.

Finally, we give some intuition on how byzantine servers can influence $\mathcal{G}$ and thus the interpretation of $\mathcal{P}$. When running gossip, a byzantine server $\check{s}$ can only manipulate the state of $\mathcal{G}$ by (1) sending an equivocating block, *i.e.* building a B and B' with $\check{s} = B.\mathsf{parent.n}$ and $\check{s} = B'.\mathsf{parent.n}$. When interpreting B and B', s will split the state for $\check{s}$ and have two 'versions' of $\mathsf{PIs}[\ell_j]$—$B'.\mathsf{PIs}[\ell_j]$ and $B.\mathsf{PIs}[\ell_j]$—sending conflicting messages for ℓ_j to servers referencing B and B'. But as $\mathcal{P}$ is a BFT protocol, the servers s_i simulating $\mathcal{P}$ (run by s) can deal with equivocation. Then $\check{s}$ could (2) reference a block multiple times, or (3) never reference a block. But again as $\mathcal{P}$ is a BFT protocol, the servers s_i simulating $\mathcal{P}$ can deal with duplicate messages and with silent servers.

Going back to Algorithm 2, the key task of s interpreting $\mathcal{G}$ is to get messages from one block to the next block. So we can see this interpretation of a block DAG as an implementation of a *communication channel*. That is, for a correct server s executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ *(i)* a server s_1 *sends* messages $m_1, \ldots, m_k$ for a protocol instance ℓ_j in either line 6 or line 11 of Algorithm 2, and *(ii)* a server s_2 *receives* a message m for a protocol instance ℓ_j in line 11 of Algorithm 2. The next lemma relates the sent and received messages with the message buffers Ms and follows from tracing changes to the variables in Algorithm 2:

Lemma 4.1. *For a correct server s executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$*

(1) a server s_1 sends *m for a protocol instance ℓ' iff there is a $B_1 \in \mathcal{G}$ with $B_1.\mathsf{n} = s_1$ such that $m \in B_1.\mathsf{Ms}[\mathsf{out}, \ell']$ for a $B' \in \mathcal{G}$ with $(\ell', r) \in B'.\mathsf{rs}$ and $B' \rightharpoonup^* B_1$.*

(2) a server s_2 receives *a message m for protocol instance ℓ' iff there are some $B_1, B_2 \in \mathcal{G}$ with $B_1 \rightharpoonup B_2$ and $B_2.\mathsf{n} = s_2$ and $m \in B_2.\mathsf{Ms}[\mathsf{in}, \ell']$ for a $B' \in \mathcal{G}$ such that $(\ell', r) \in B'.\mathsf{rs}$ and $B' \rightharpoonup^* B_1$.*

The following lemma shows our key observation from before: interpreting a block DAG is independent from the server doing the interpretation. That is, s and s' will arrive at the same state when interpreting $B \in \mathcal{G}$.

Lemma 4.2. *If $\mathcal{G} \leqslant \mathcal{G}'$ then for every $B \in \mathcal{G}$, a deterministic protocol $\mathcal{P}$ and correct servers s and s' executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ and $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$ it holds that $B.\mathsf{PIs}[\ell_j] = B.\mathsf{PIs}'[\ell_j]$ and $B.\mathsf{Ms}[\mathsf{out}, \ell_j] = B.\mathsf{Ms}'[\mathsf{out}, \ell_j]$ for $(\ell_j, r) \in B_j.\mathsf{rs}$ with $B_j \rightharpoonup^n B$ for $n \geqslant 0$.*

Proof. In this proof, when executing $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$ we write Ms' and PIs' to distinguish from Ms and PIs when executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$. We show $B_1.\mathsf{Ms}[\mathsf{out}, \ell_j] = B_1.\mathsf{Ms}'[\mathsf{out}, \ell_j]$ and $B_1.\mathsf{PIs}[\ell_j] = B_1.\mathsf{PIs}'[\ell_j]$ by induction on n—the length of the path from B_j to B_1 in $\mathcal{G}$ and $\mathcal{G}'$. For the base case we have $B_1 = B_j$ and $\ell_j \in \{\ell_j \mid (\ell_j, r_j) \in B_1.\mathsf{rs}\}$. By [22, Lemma A.10], B_1 is picked eventually in line 3 of Algorithm 2 when executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$. Then, by line 6 $B_1.\mathsf{Ms}[\mathsf{out}, \ell]$ is $B_1.\mathsf{PIs}[\ell_j].(B_1.\mathsf{rs})$. By the same reasoning, when executing $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$, $B_1.\mathsf{Ms}'[\mathsf{out}, \ell] = B_1.\mathsf{PIs}[\ell_j].(B_1.\mathsf{rs})$. As $B_1.\mathsf{PIs}[\ell_j].(B_1.\mathsf{rs})$ are deterministic and depend only on B_1, ℓ_j, and $\mathcal{P}$, we know that $B_1.\mathsf{PIs}[\ell] = B_1.\mathsf{PIs}'[\ell]$ and $B_1.\mathsf{PIs}[\ell] = B_1.\mathsf{PIs}'[\ell]$, and conclude the base case. For the step case by induction hypothesis for $B_i \in B_1.\mathsf{preds}$ with $B_j \rightharpoonup^{n-1} B_i$ holds *(i)* $B_i.\mathsf{Ms}[\mathsf{out}, \ell_j] = B_i.\mathsf{Ms}'[\mathsf{out}, \ell_j]$, and *(ii)* $B_i.\mathsf{PIs}[\ell_j] = B_i.\mathsf{PIs}'[\ell_j]$. Again by [22, Lemma A.10], B_1 is picked eventually in line 3 of Algorithm 2 when executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ and $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$. In line 4 and as $B_1.\mathsf{parent} \in B_1.\mathsf{preds}$ and *(ii)*, now $B_1.\mathsf{PIs}[\ell_j] = B_1.\mathsf{PIs}'[\ell_j]$. Now, as $\mathcal{P}$ is deterministic, we only need to establish that $B_1.\mathsf{Ms}[\mathsf{in}, \ell_j] = B_1.\mathsf{Ms}'[\mathsf{in}, \ell_j]$ to conclude that $B_1.\mathsf{PIs}[\ell_j] = B_1.\mathsf{PIs}'[\ell_j]$ and $B_1.\mathsf{Ms}[\mathsf{out}, \ell_j] = B_1.\mathsf{Ms}'[\mathsf{out}, \ell_j]$, which as $(\ell_j, r) \notin$

[1] An equivalent representation would keep process instances $\mathsf{PIs}[B, \ell_j, B.\mathsf{n}]$ and message buffers $\mathsf{Ms}[B, d \in \{\mathsf{in}, \mathsf{out}\}, \ell_j]$ explicitly as global state. We chose this notation to accentuate the information flow throughout $\mathcal{G}$.

B_1.rs, is only modified in this line 11. By [22, Lemma A.9], we know for both executions that $B_1.\mathsf{Ms}[\mathsf{in}, \ell_j] = B_1.\mathsf{Ms}'[\mathsf{in}, \ell_j] = \emptyset$, before B_1 is picked. Now, by *(i)* and line 9 $B_1.\mathsf{Ms}[\mathsf{in}, \ell_j] = B_1.\mathsf{Ms}'[\mathsf{in}, \ell_j]$, and we conclude the proof. □

A straightforward consequence of Lemma 4.2 is, that when in the interpretation of s, a server s_1 sends a message m for ℓ_j, then s_1 sends m in the interpretation of s' [22, Lemma A.16]. Curiously, s_1 does not have to be correct: we know s_1 sent a block B in $\mathcal{G}$, that corresponds to a message m in the interpretation of s. Now this block will be interpreted by s' and the same message will be interpreted—and for that the server s_1 does not need to be correct. By Lemma 4.3 $\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ has the properties of an authenticated perfect point-to-point link after [3, Module 2.5, p. 42].

Lemma 4.3. *For a block DAG $\mathcal{G}$ and a correct server s executing $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ holds*

*(1) if a correct server s_1 sends a message m for a protocol instance ℓ to a correct server s_2, then s_2 eventually receives m for protocol instance ℓ for a correct server s' executing $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$ and a block DAG $\mathcal{G}' \geqslant \mathcal{G}$ (*reliable delivery*).*

*(2) for a protocol instance ℓ no message is received by a correct server s_2 more than once (*no duplication*).*

*(3) if some correct server s_2 receives a message m for protocol instance ℓ with sender s_1 and s_1 is correct, then the message m for protocol instance ℓ was previously sent to s_2 by s_1 (*authenticity*).*

Proof Sketch. For (1), we observe that every message m sent in $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ will be sent in $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$ for $\mathcal{G}' \geqslant \mathcal{G}$ by [22, Lemma A.16]. Now by Lemma 3.7, s' will eventually have some $\mathcal{G}' \geqslant \mathcal{G}$. By Lemma 4.1 (1) we have witnesses $B_1, B_2 \in \mathcal{G}'$ with $B_1 \rightharpoonup B_2$, and by Lemma 4.1 (2) we found a witness B_2 to receive the message on when executing $s'.\mathsf{interpret}(\mathcal{G}', \mathcal{P})$. For (2), we observe, that duplicate messages are only possible if s_2 inserted the block B_1, which gives rise to the message m, in two different blocks built by s_2. But this contradicts the correctness of s_2 by [22, Lemma A.6]. For (3), we observe that only s_1 can build and sign any block B_1 with $s_1 = B.\mathsf{n}$, which gives rise to m. □

Before we compose gossip and interpret in the next section under a shim, we highlight the key benefits of using interpret in Algorithm 2. By leveraging the block DAG structure together with $\mathcal{P}$'s determinism, we can *compress messages* to the point of omitting some of them. When looking at line 11 of Algorithm 2, the messages in the buffers $\mathsf{Ms}[\mathsf{out}, \ell]$ and $\mathsf{Ms}[\mathsf{in}, \ell]$ have never been sent over the network. They are locally computed, functional results of the calls receive(m). The only 'messages' actually sent over the network are the requests r_i read from B.rs in line 6. To determine the messages following from these request, the server s simulates an instance of protocol $\mathcal{P}$ for every $s_i \in \mathsf{Srvrs}$—simply by simulating the steps in the deterministic protocol. However, not every step can be simulated: as s does not know s_i's private key, s cannot sign a message on s_i's behalf. But then, this is not necessary, because s can derive the authenticity of the message triggered by a block B from the signature of B, *i.e.*, $B.\sigma$. So instead of signing individual messages, s_i can give a *batch signature* $B.\sigma$ for authenticating every message materialized through B. Finally, s interprets protocol

```
1  module shim(s ∈ Srvrs, P ∈ module)
2      rqsts := ∅ ∈ 2^(L×Rqsts)
3      G := ∅ ∈ Dags
4      gssp := new process gossip(s, G, rqsts)
5      intprt := new process interpret(G, P)
6      when request(ℓ ∈ L, r ∈ Rqsts)
7          rqsts.put(ℓ, r)
8      when intprt.indicate(ℓ, i, s') where s' = s
9          indicate(ℓ, i)
10     repeatedly
11         gssp.disseminate()
```

Algorithm 3: Interfacing between gossip, interpret and user of $\mathcal{P}$.

instances with labels ℓ_j *in parallel* in line 7 of Algorithm 2. While traversing the block DAG, s uses the structure of the block DAG to interpret requests and messages for every ℓ_j. Now, the same block giving rise to a request in process instance ℓ_j may materialize a message in process instance ℓ'_j. The (small) price to pay is the increase of block size by references to predecessor blocks, *i.e.*, B.preds. We will illustrate the benefits again on the concrete example of byzantine reliable broadcast in the next Section 5.

5 USING THE FRAMEWORK

The protocol $\mathsf{shim}(\mathcal{P})$ in Algorithm 3 is responsible for the choreography of the external user of $\mathcal{P}$, the gossip protocol in Algorithm 1, and the interpret protocol in Algorithm 2. Therefore, the server s executing $\mathsf{shim}(\mathcal{P})$ in Algorithm 3 keeps track of two synchronized data structures (1) a buffer of labels and requests rqsts in line 2, and (2) and the block DAG $\mathcal{G}$ in line 3. By calling $\mathsf{rqsts.put}(\ell, r)$, s inserts (ℓ, r) in rqsts, and by calling $\mathsf{rqsts.get}()$, s gets *and* removes a suitable number of requests $(\ell_1, r_1), \ldots, (\ell_n, r_n)$ from rqsts. To insert a block B in $\mathcal{G}$, s calls $\mathcal{G}.\mathsf{insert}(B)$ from Definition 3.4. We tacitly assume these operations are atomic. When starting an instance of gossip and interpret in line 4 and 5, s passes in references to theses shared data structures. When the external user of protocol $\mathcal{P}$ requests $r \in \mathsf{Rqsts}$ for $\ell \in \mathcal{L}$ from s via the request $\mathsf{request}(\ell, r)$ to $\mathsf{shim}(\mathcal{P})$ then s inserts (ℓ, r) in rqsts in lines 6–7. By executing gossip, s writes (ℓ, r) in $\mathcal{B}$ in Algorithm 1 line 15, and as eventually $\mathcal{B} \in \mathcal{G}$, r will be requested from protocol instance $\mathsf{PIs}[\ell]$ when s executes line 6 in Algorithm 2 [22, Lemma A.17]. On the other hand, when interpret indicates $i \in \mathsf{Inds}$, for the interpretation of $\mathcal{P}$ *for itself, i.e.*, $s = s'$, then s indicates to the user of $\mathcal{P}$ in line 8–9 of Algorithm 3 [22, Lemma A.18]. For s to only indicate when $s = s'$ might be an over-approximation: s trusts s's interpretation of $\mathcal{P}$ as s is correct for s. We believe this restriction can be lifted (*cf.* Section 7). Finally, as promised in Section 3, in lines 10–11 s repeatedly requests disseminate from gossip to disseminate $\mathcal{B}$. Within the control of s, the time between calls to disseminate can be adapted to meet the network assumptions of $\mathcal{P}$ and can be enforced *e.g.* by an internal timer, the block's payload, or when s falls n blocks behind. For our proofs we only need to guarantee that a correct s will eventually request disseminate.

Following [3], a protocol $\mathcal{P}$ implements an interface $\mathbb{I}$ and has properties $\mathbb{P}$, which are shown to hold for $\mathcal{P}$. For any property, which holds for a protocol $\mathcal{P}$ and where the proof of the property relies on the reliable point-to-point abstraction in Lemma 4.3, $\mathbb{P}$ holds for $\mathsf{shim}(\mathcal{P})$. Again following [3], these are the properties of any algorithm that *uses* the reliable point-to-point link abstraction.

Taking together what we have established for gossip in Section 3, *i.e.* that correct servers will eventually share a joint block DAG, and that interpret gives a point-to-point link between them in Section 4, for $\mathsf{shim}(\mathcal{P})$ the following holds:

Theorem 5.1. *For a correct server s and a deterministic protocol $\mathcal{P}$, if $\mathcal{P}$ is an implementation of* (i) *an interface $\mathbb{I}$ with requests $\mathsf{Rqsts}_{\mathcal{P}}$ and indications $\mathsf{Inds}_{\mathcal{P}}$ using the reliable point-to-point link abstraction such that* (ii) *a property $\mathbb{P}$ holds, then $\mathsf{shim}(\mathcal{P})$ in Algorithm 3 implements* (i) *$\mathbb{I}$ such that* (ii) *$\mathbb{P}$ holds.*

Proof. By [22, Lemma A.17] and [22, Lemma A.18], *(i)* $\mathsf{shim}(\mathcal{P})$ implements the interface $\mathbb{I}$ of $\mathsf{Rqsts}_{\mathcal{P}}$ and $\mathsf{Inds}_{\mathcal{P}}$. For *(ii)*, by assumption $\mathbb{P}$ holds for $\mathcal{P}$ using a reliable point-to-point link abstraction. By Lemma 4.3 $s.\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ implements a reliable point-to-point link. As Algorithm 2 treats $\mathcal{P}$ as a black-box every $B.\mathsf{PIs}[\ell]$ holds an execution of $\mathcal{P}$. Assume this execution violates $\mathbb{P}$. But then an execution of $\mathcal{P}$ violates $\mathbb{P}$ which contradicts the assumption that $\mathbb{P}$ holds for $\mathcal{P}$. □

Our proof relies on a point-to-point link between two correct servers and thus we can translate the argument of all safety and liveness properties, for which their reasoning relies on the point-to-point link abstraction, to our block DAG framework. Because we provide an abstraction, we cannot directly translate implementation-level properties measuring performance such as latency or throughput. They rely on the concrete implementation. Also, as discussed in Section 4, properties related to signatures do not directly translate, because blocks—not messages—are (batch-)signed.

While our work focuses on correctness, two recent works show that DAG-based approaches for concrete protocols $\mathcal{P}$ are efficient and even optimal: DAG-Rider [16] implements the asynchronous Byzantine Atomic Broadcast abstraction and is shown to be optimal with respect to resilience, amortized communication complexity, and time. Different to our work, DAG-Rider relies on randomness, which is an extension in our setting. On the other hand Narwhal and Tusk [8] for BFT consensus do not rely on randomness and report impressive—also empirically evaluated—performance gains. Moreover, as argued in [8], our approach enjoys two further benefits for implementations: load balancing, as we do not rely on a single leader, and equal message size. Finally, we note that in our setting the complexity measure of calls to the reliable point-to-point link abstraction is slightly misleading, because we are optimizing the messages transmitted by the point-to-point link abstraction.

$\mathcal{P} :=$ *byzantine reliable broadcast.* In the remainder of this section, we will sketch how a user may use the block DAG framework. Our example for $\mathcal{P}$ is *byzantine reliable broadcast* (BRB)—a protocol underlying recently-proposed efficient payment systems [2, 13]. Given an implementation of byzantine reliable broadcast after [3, Module 3.12, p. 117], *e.g.*, [22, Algorithm 4]: this is the $\mathcal{P}$, which the user passes to $\mathsf{shim}(\mathcal{P})$, *i.e.* in the block DAG framework $\mathcal{P}$ is fixed to an implementation of BRB, *e.g.*, [22, Algorithm 4]. The request in BRB is $\mathsf{broadcast}(v)$ for a value $v \in \mathsf{Vals}$, so $\mathsf{Rqsts}_{\mathcal{P}} = \{\mathsf{broadcast}(v) \mid v \in \mathsf{Vals}\}$. For simplicity and generality, we assume that $\mathcal{P}$—not $\mathsf{shim}(\mathcal{P})$—authenticates requests, *i.e.* requests are self-contained and can be authenticated while simulating $\mathcal{P}$ (*e.g.*, [22, Algorithm 4, line 3]). However, in an implementation $\mathsf{shim}(\mathcal{P})$ may be employed to authenticate requests. On the other hand, BRB indicates with $\mathsf{deliver}(v)$, so $\mathsf{Inds}_{\mathcal{P}} = \{\mathsf{deliver}(v) \mid v \in \mathsf{Vals}\}$. The messages sent in BRB are $\mathsf{M}_{\mathcal{P}} = \{\mathtt{ECHO}\ v, \mathtt{READY}\ v \mid v \in \mathsf{Vals}\}$ where sender and receiver are the $s \in \mathsf{Srvrs}$ running $\mathsf{shim}(\mathcal{P})$. When executing line 9 of $\mathsf{interpret}(\mathcal{G}, \mathcal{P})$ in Algorithm 2, then $\mathsf{receive}(\mathtt{ECHO}\ 42)$ is triggered, and **received** ECHO 42 holds (*e.g.*, [22, Algorithm 4, line 6]). As we assume $\mathcal{P}$ returns messages immediately, *e.g.*, when the simulation reaches **send** ECHO 42, then ECHO 42 is returned immediately (*e.g.*, [22, Algorithm 4, line 8]). The interface $\mathbb{I}$ is $\mathsf{Rqsts} = \{\mathsf{broadcast}(v) \mid v \in \mathsf{Vals}\}$ and $\mathsf{Inds} = \{\mathsf{deliver}(v) \mid v \in \mathsf{Vals}\}$ The properties $\mathbb{P}$ of BRB—validity, no duplication, integrity, consistency, and totality—are preserved.

Figure 4 shows a block DAG for an execution of $\mathsf{shim}(P)$ using byzantine reliable broadcast. It further explicitly shows the in- and out-going messages from $\mathsf{Ms}[\mathsf{in}, \ell_1]$ and $\mathsf{Ms}[\mathsf{out}, \ell_1]$ for a protocol instance ℓ_1 and the request $\mathsf{broadcast}(42)$ at block B_1. None of these messages are ever actually sent over the network—every server interpreting this block DAG can use interpret in Algorithm 2 to replay an implementation of BRB, *e.g.* [22, Algorithm 4], and get the same picture. Figure 4 shows only the (unsent) messages for ℓ_1 and $\mathsf{broadcast}(42))$ in $B_1.\mathsf{rs}$, but $B_1.\mathsf{rs}$ may hold more requests such as $\mathsf{broadcast}(21)$ for ℓ_2, and all the messages of all these requests could be materialized in the same manner—without any messages, or even additional blocks, sent. And not only B_1 holds such requests—also B_3 does. For example, $B_3.\mathsf{rs}$ may contain $\mathsf{broadcast}(25)$ for ℓ_3. Then, for ℓ_3 on B_3 materializes $\mathsf{out} = \mathtt{ECHO}\ 25$ to s_1, s_2, s_3, and again, without sending any messages, for ℓ_3 on B_6, B_7, and B_8 materializes $\mathsf{in} = \mathtt{ECHO}\ 25$ from s_2. This is, of course, the same for every B_i.

To recap, what makes interpreting $\mathcal{P}$ on a block DAG so attractive: sending blocks instead of messages in a deterministic $\mathcal{P}$ results in a compression of messages—up to their omission. And not only do these messages not have to be sent, they also do not have to be signed. It suffices, that every server signs their blocks. Finally, a single block sent is interpreted as messages for a very large number of parallel protocol instances.

6 RELATED WORK

The last years have seen many proposals based on block DAG paradigms (see [23] for an SoK)—some with commercial implementations. We focus on the proposals closest to our work: Hashgraph [1], Blockmania [7], Aleph [12], and Flare [21]. Underlying all of these systems is the same idea: first, build a common block DAG, and then locally interpret the blocks and graph structure as communication for some protocol: Hashgraph encodes a consensus protocol in block DAG structure, Blockmania [7] encodes a simplified version of PBFT [4], Aleph [12] employs atomic broadcast and consensus, and Flare [21] builds on federated byzantine agreement from Stellar [20] combined with block DAGs to implement a federated voting protocol. Naturally, the correctness arguments of these systems focus on their system, *e.g.*, the correctness proof in Coq of byzantine consensus in Hashgraph [6]. In

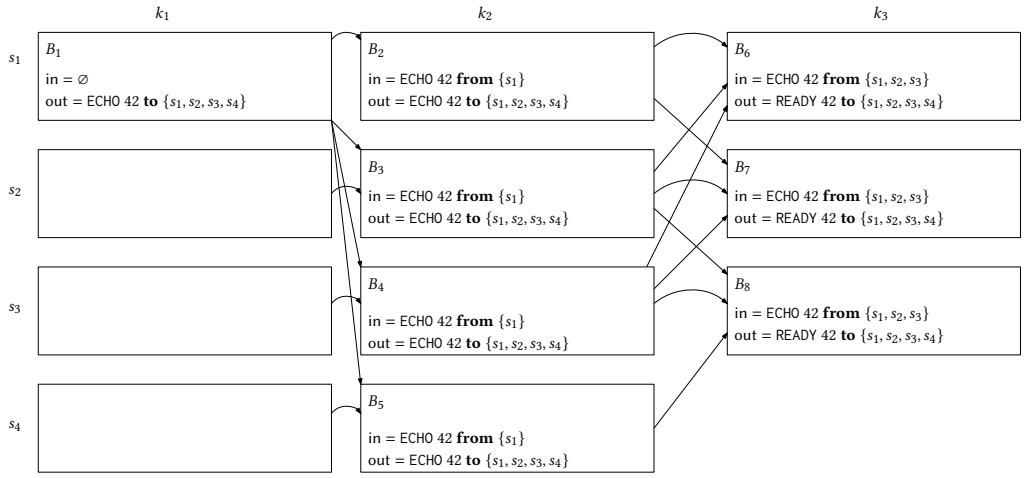

Figure 4: The message buffers for process instance ℓ_1 of a block DAG with $(\ell_1, \text{broadcast}(42)) \in B_1.\text{rs}$

our work, we aim for a different level of generality: we establish structure underlying protocols which employ block DAGs, *i.e.*, we show that a block DAG implements a reliable point-to-point channel (Section 4). To that end, and opposed to previous approaches, we treat the protocol $\mathcal{P}$ completely as a black-box, *i.e.*, our framework is parametric in the protocol $\mathcal{P}$.

The idea to leverage deterministic state machines to replay the behavior of other servers goes back to PeerReview [14], where servers exchange logs of received messages for auditing to eventually detect and expose faulty behavior. This idea was taken up by block DAG approaches—but with the twist to leverage determinism to *not* send those messages that can be determined. This allows compressing messages to the extent of only indicating that a message has been sent as we do in Section 4. However, we believe nothing precludes our proposed framework to be adapted to hold equivocating servers accountable, drawing *e.g.*, on recent work from Polygraph to detect byzantine behavior [5].

While our framework treats the interpreted protocol $\mathcal{P}$ as a black-box, the recently proposed threshold logical clock abstraction [11] allows the higher-level protocol to operate on an asynchronous network as if it were a synchronous network by abstracting communication of groups. Similar to our framework, also threshold clocks rely on causal relations between messages by including a threshold number of messages for the next time step. This would roughly correspond to including a threshold number of predecessor blocks. In contrast, our framework, by only providing the abstraction of a reliable point-to-point link to $\mathcal{P}$, pushes reasoning about messages to $\mathcal{P}$.

7 EXTENSIONS, LIMITATIONS & CONCLUSION

We have presented a generic formalization of a block DAG and its properties, and in particular results relating to the eventual delivery of all blocks from correct servers to other correct servers. We then leverage this property to provide a concrete implementation of a reliable point-to-point channel, which can be used to implement any deterministic protocol $\mathcal{P}$ efficiently. In particular we have efficient message compression, as those messages emitted by $\mathcal{P}$, which are the results of the deterministic execution of $\mathcal{P}$ may be omitted. Moreover we are allowing for batching of the execution of multiple parallel instances of $\mathcal{P}$ using the same block DAG, and the decoupling of maintaining the joint block DAG from its interpretation as instances of $\mathcal{P}$.

Extensions. First, throughout our work we assume $\mathcal{P}$ is deterministic. The protocol may accept user requests, and emit deterministic messages based on these events and other messages. However, it may not use any randomness in its logic. It seems we can extend the proposed composition to non-deterministic protocols $\mathcal{P}$—but some care needs to be applied around the security properties assumed from randomness. In case randomness is merely at the discretion of a server running their instance of the protocol we can apply techniques to de-randomize the protocol by relying on the server including in their created block any coin flips used. In case randomness has to be unbiased, as is the case for asynchronous Byzantine consensus protocols, a joint shared randomness protocol needs to be embedded and used to de-randomize the protocol. Luckily, shared coin protocols that are secure under BFT assumptions and in the synchronous network setting exist [17] and our composition could be used to embed them into the block DAG. However we leave the details of a generic embedding for non-deterministic protocols for future work.

Second, we have discussed the case of embedding asynchronous protocols into a block DAG. We could extend this result to BFT protocols in the partial synchronous network setting [9] by showing that the block DAG interpretation not only creates a reliable point-to-point channel but also that its delivery delay is bounded if the underlying network is partially synchronous. We have a proof sketch to this effect, but a complete proof would require to introduce machinery to reason about timing and, we believe, would not enhance the presentation of the core arguments behind our abstraction.

Third, our correctness conditions on the block DAG seem to be much more strict than necessary. For example, block validity requires a server to have processed all previous blocks. In practice

this results in blocks that must include at some position k all predecessors of blocks to be included after position k. This leads to inefficiencies: a server must include references to all blocks by other parties into their own blocks, which represents an $O(n^2)$ overhead (admittedly with a small constant, since a cryptographic hash is sufficient). Instead, block inclusion could be more implicit: when a server s includes a block B' in its block B all predecessors of B' could be implicitly included in the block B, transitively or up to a certain depth. This would reduce the communication overhead even further. Since it is possible to take a block DAG with this weaker validity condition and unambiguously extract a block DAG with the stronger validity condition we assume, we foresee no issues for all our theorems to hold. Furthermore, when interpreting a protocol currently a server only indicates, when the server running the interpretation indicates in the interpretation. This is to assure that the server running the interpretation can trust the server in the interpretation, *i.e.* itself. Again, we believe that this can be weakened by leveraging properties of the interpreted protocol. However, we again leave a full exploration of this space to future work.

Limitations. Some limitations of our composition require much more foundational work to be overcome. And these limitations also apply to the block DAG based protocols which we attempt to formalize. First, there are practical challenges when embedding protocols tolerating processes that can crash and recover. At first glance safe protocols in the crash recovery setting seem like a great match for the block DAG approach: they do allow parties that recover to re-synchronize the block DAG, and continue execution, assuming that they persist enough information (usually in a local log) as part of $\mathcal{P}$. However there are challenges: first, our block DAG assumes that blocks issued have consecutive numbers. If the higher-level protocols use these block sequence numbers as labels for state machines (as in Blockmania), a recovering process may have to 'fill-in' a large number of blocks before catching up with others. An alternative is for block sequence numbers to not have to be consecutive, but merely increasing, which would remove this issue.

However in all cases, unless there is a mechanism for the higher level protocol $\mathcal{P}$ to signal that some information will never again be needed, the full block DAG has to be stored by all correct parties forever. This seems to be a limitation of both our abstraction of block DAG but also the traditional abstraction of reliable point-to-point channels and the protocols using them, that seem to not require protocols to ever signal that a message is not needed any more (to stop re-transmission attempt to crashed or Byzantine servers). Fixing this issue, and proving that protocols can be embedded into a block DAG, that can be operated and interpreted using a bounded amount of memory to avoid exhaustion attacks is a challenging and worthy future avenue for work – and is likely to require a re-thinking of how we specify BFT protocols in general to ensure this property, beyond their embedding into a block DAG.

Finally, one of the advantages of using a block DAG is the ability to separate the operation and maintenance of the block DAG from the later or off-line interpretation of instances of protocol $\mathcal{P}$. However, this separation does not hold and extend to operations that change the membership of the server set that maintain the block DAG—often referred to as reconfiguration. How to best support reconfiguration of servers in block DAG protocols seems to be an open issue, besides splitting protocol instances in pre-defined epochs.

ACKNOWLEDGMENTS

This work has been partially supported the UK EPSRC Project EP/R006865/1, Interface Reasoning for Interacting Systems (IRIS).

REFERENCES

[1] Leemon Baird. 2016. *The Swirlds Hashgraph Consensus Algorithm: Fair, Fast, Byzantine Fault Tolerance.* Technical Report. 28 pages.
[2] Mathieu Baudet, George Danezis, and Alberto Sonnino. 2020. FastPay: High-Performance Byzantine Fault Tolerant Settlement. arXiv:2003.11506
[3] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (second ed.). Springer-Verlag, Berlin Heidelberg.
[4] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, Berkeley, CA, USA, 173–186.
[5] Pierre Civit, Seth Gilbert, and Vincent Gramoli. 2020. Brief Announcement: Polygraph: Accountable Byzantine Agreement. In *34th International Symposium on Distributed Computing (DISC 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 179)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 45:1–45:3. https://doi.org/10.4230/LIPIcs.DISC.2020.45
[6] Karl Crary. 2021. Verifying the Hashgraph Consensus Algorithm. arXiv:2102.01167
[7] George Danezis and David Hrycyszyn. 2018. Blockmania: From Block DAGs to Consensus. arXiv:1809.01620
[8] George Danezis, Eleftherios Kokoris Kogias, Alberto Sonnino, and Alexander Spiegelman. 2021. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. arXiv:2105.11827
[9] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35 (1988), 288–323.
[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382.
[11] Bryan Ford. 2019. Threshold Logical Clocks for Asynchronous Distributed Coordination and Consensus. arXiv:1907.07010
[12] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies (AFT '19)*. ACM, New York, NY, USA, 214–228. https://doi.org/10.1145/3318041.3355467
[13] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2019. The Consensus Number of a Cryptocurrency (Extended Version). In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*. ACM, New York, NY, USA, 307–316. https://doi.org/10.1145/3293611.3331589
[14] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of the Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 175–188. https://doi.org/10.1145/1294261.1294279
[15] Jonathan Katz and Yehuda Lindell. 2007. *Introduction to Modern Cryptography.* Chapman & Hall/CRC.
[16] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need Is DAG. arXiv:2102.08325
[17] 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 (CCS '20)*. ACM, New York, NY, USA, 1751–1767. https://doi.org/10.1145/3372297.3423364
[18] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563
[19] Petros Maniatis and Mary Baker. 2002. Secure History Preservation Through Timeline Entanglement. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 297–312.
[20] David Mazières. 2015. *The Stellar Consensus Protocol: A Federated Model for Internet-Level Consensus.* Technical Report.
[21] Sean Rowan and Naïri Usher. 2019. The Flare Consensus Protocol: Fair, Fast Federated Byzantine Agreement Consensus.
[22] Maria A Schett and George Danezis. 2021. Embedding a Deterministic BFT Protocol in a Block DAG. arXiv:2102.09594
[23] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. 2020. SoK: Diving into DAG-Based Blockchain Systems. arXiv:2012.06128