Faithful Simulation of Randomized BFT Protocols on Block DAGs

Hagit Attiya
Technion, Israel

Constantin Enea
LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris, France

Shafik Nassar
Technion, Israel

Abstract

Byzantine Fault-Tolerant (BFT) protocols that are based on *Directed Acyclic Graphs* (DAGs) are attractive due to their many advantages in asynchronous blockchain systems. These DAG-based protocols can be viewed as a *simulation* of some BFT protocol on a DAG. Many DAG-based BFT protocols rely on randomization, since they are used for agreement and ordering of transactions, which cannot be achieved deterministically in asynchronous systems. Randomization is achieved either through local sources of randomness, or by employing shared objects that provide a common source of randomness, e.g., *common coins*.

A DAG simulation of a randomized protocol should be *faithful*, in the sense that it precisely preserves the properties of the original BFT protocol, and in particular, their probability distributions. We argue that faithfulness is ensured by a *forward simulation*. We show how to faithfully simulate any BFT protocol that uses public coins and shared objects, like common coins.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Distributed computing models; Computing methodologies → Distributed algorithms; General and reference → Verification

Keywords and phrases Distributed Algorithms, Byzantine failures, Hyperproperties, Forward Simulation

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.27

Related Version https://eprint.iacr.org/2023/192

Funding *Hagit Attiya*: partially supported by the Israel Science Foundation (grants 380/18 and 22/1425).
Constantin Enea: partially supported by the project AdeCoDS of the French ANR Agency.
Shafik Nassar: partially funded by the European Union (ERC, FASTPROOF, 101041208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

Asynchronous distributed computation is naturally captured by a *directed acyclic graph* (DAG), whose nodes describe local computation and edges correspond to causal dependency between computation at different processes. Lamport's *happens-before* relation [14] is an example of such DAG, where each node is a single local computation event, and each edge is a single message delivery event. *Block* DAGs [21] go one step further and incorporate more than one local computation step in each block (node); these steps may even belong to several *independent* protocols.

By exchanging blocks in a manner that preserves their dependencies, a distributed protocol can now be abstracted as a joint computation of a block DAG. In particular, a general *Byzantine fault-tolerant* (BFT) DAG-based algorithm combines two components: one component builds the DAG using a communication protocol that tolerates malicious failures, and the other component performs the local computation embodied in each node of the DAG. The first component can be used to separate the task of injecting user input to the system, such as transactions, from the task of processing these inputs and producing an output, e.g., an ordering of those transactions.

This generality makes block DAGs an attractive approach for designing coordination protocols for, e.g., Byzantine Atomic Broadcast [10,13,20], consensus [4,16] and cryptocurrencies [6]. (For a survey of the techniques used in block DAG approaches, see [21].) A block DAG can be seen as a strict extension of a *blockchain*, which is a DAG where all blocks are *totally ordered*, i.e., a directed path. The DAG approach was shown to achieve high throughput [19] due to the flexibility it provides over the standard blockchain approach.

Schett and Danezis [17] show that any *deterministic* BFT protocol can be simulated as a block DAG. They provide generic mechanisms for processes to maintain a consistent view of the block DAG, and to individually *interpret* the DAG as an execution of some protocol.

The restriction to deterministic protocols, however, handicaps the applicability of this result, since many algorithms in the asynchronous domain are necessarily non-deterministic, due to the FLP impossibility result [9]. For example, DAG-based agreement protocols with provable security, like Aleph [10] or DAG-Rider [13], are either randomized or assume the existence of a shared source of randomness. This calls for a framework that can handle *randomized* BFT protocols; those that either utilize local randomness or even a shared object.

The problem of using or defining block DAG simulations in the context of *randomized protocols* has two aspects: (1) using a block DAG simulation of a *deterministic* protocol as a building block of a *randomized protocol*, and (2) defining block DAG simulations of *randomized protocols*.

Concerning the first aspect above, we aim to enable modular reasoning when using such simulations instead of the original protocols (Section 2 describes a concrete example). Schett and Danezis [17] establish that the traces of the block DAG simulation are included in the set of traces of the original protocol (for some notion of trace which is not important for this discussion). However, as shown in other contexts, e.g., concurrent objects [2,11], such a notion of refinement is not sufficient to conclude that relevant specifications of a randomized protocol that builds on some other deterministic protocol are preserved when the latter is replaced by the block DAG simulation. Indeed, the specifications of randomized protocols characterize sets (probabilistic distributions) of executions and are instances of *hyper-properties* which are not preserved by standard trace inclusion [2].

Therefore, we establish a stronger notion of refinement between a block DAG simulation and the original protocol, namely, that there exists a *forward simulation* between the two. (A forward simulation maps every step of one protocol to a sequence of steps of the other protocol, starting from the initial state of the first and advancing in a forward manner; a backward simulation is similar, but it goes in the reverse direction, from end states back to initial states.). Based on the results in [2], this implies that any finite-trace specification of a randomized protocol against an adaptive adversary is preserved when a sub-protocol is replaced by its block DAG simulation. We recall that an *adaptive adversary* is a scheduler that resolves all the non-determinism introduced by the interleaving semantics and which can observe everything about the local state of a process or the messages in transit.

Armed with this understanding of the precise nature of block DAG simulation, we present

Algorithm 1 Binary consensus using a common coin

Input: x

$r := 0$; $est := x$;
while *true* **do**
    r++;
    $val := r.\mathsf{BCA}(est)$;
    $c := r.\mathsf{Toss}()$;
    if $val \neq \perp$ and $c = val$ **then**
        output val;
        $est := val$;
    else if $val \neq \perp$ **then**
        $est := val$;
    else
        $est := c$;

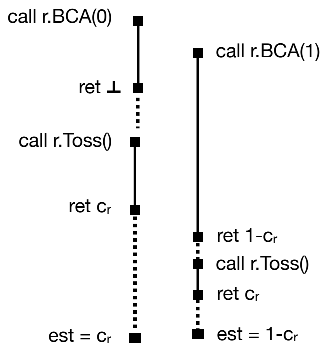

Figure 1 A randomized consensus algorithm on the left, and an execution template ($c_0 \in \{0, 1\}$) on the right, which represents the executions of an adaptive adversary which disallows termination.

an extension of the construction of Schett and Danezis [17], which applies also to protocols using randomization and shared objects. Specifically, we consider *randomized* protocols in which the local coin flips of each process may be public, we call those protocols *public-coin* protocols. We prove that any public-coin protocol that uses shared objects, e.g., common coins, can be simulated on a block DAG, preserving its usage of shared objects.

A relationship based on a forward simulation allows to conclude that probabilistic specifications of a randomized protocol, e.g., termination time, are preserved by its block DAG simulation. Such a simulation precisely preserves the finite trace distribution and the probabilistic relationship between inputs and outputs. This means that whatever "adverse" effects can occur in the simulation, can already be demonstrated in the original protocol.

Organization. Section 2 presents an example that demonstrates why simulations should preserve hyperproperties. Sections 3–5 describe the model and introduces important definitions and notations. Section 6 formally defines block DAGs. Our results are presented and proved in Section 7. The relation of our simulation to [17], and some applications appear in Section 8. We summarize with future work, in Section 9.

2 Motivating Example

We describe a class of protocols solving *Binary Crusader Agreement*, and a hyperproperty about them, called *binding* [1], which is assumed when such protocols are used to solve randomized consensus. This motivates the need for establishing a notion of refinement for block DAG simulations that is stronger than trace inclusion and which enables the preservation of such hyperproperties.

2.1 Randomized consensus based on Binary Crusader Agreement

Let us consider the consensus protocol listed in Algorithm 1 (from [1]). This is a randomized protocol based on two sub-protocols, Binary Crusader Agreement, invoked as BCA, and a common coin, invoked via Toss. Every process participating in this consensus protocol goes through a sequence of asynchronous rounds (the current round is stored in the variable r), and each round consists of one instance of BCA followed by one instance of Toss. We prefix invocations with the value of r in order to emphasize that these instances are different from one round to another.

Binary Crusader Agreement [7] is a weak form of consensus, where processes start with a value in $\{0, 1\}$ and can return a value in $\{0, 1, \perp\}$ (note the special value $\perp$). The requirements are: (1) validity: if all non-faulty processes start with the same input, then this is the only output, (2) agreement: no two non-faulty processes output two distinct non-$\perp$ values, and (3) termination: every non-faulty process eventually outputs a value. It is weaker than consensus because a process can output the "don't know" value $\perp$ instead of one of the inputs. The common coin protocol allows to implement a shared source of *uniform* randomness, it guarantees that all processes receive the same output in $\{0, 1\}$ (drawn with equal probability) and that this output is unpredictable to an outsider (adversary).

Each round of the consensus protocol starts with a round of BCA where each process inputs the current estimation of the agreement value *est* (initially, this is the input x), followed by a round of the common coin. If BCA returns a non-$\perp$ value then this will be the value of *est* in the next round. Otherwise, the value of *est* is the value returned by the coin protocol. Furthermore, if the values returned by BCA and Toss are the same, then the process outputs the decision value. A process continues running the protocol after outputting the decision in order to "help" other processes reach a decision (e.g., so that future instances of BCA and the common coin satisfy honest super majority assumptions).

2.2 Termination under binding

We say that the protocol *terminates* when all non-faulty processes output a decision. It has been shown [1] that the protocol of Figure 1 terminates against an adaptive adversary with probability 1, provided that BCA satisfies a property called *binding*. The binding property states that for every execution prefix of BCA that ends with a process returning $\perp$, there is a *single* non-$\perp$ value that can be returned by a process in *any* future extension of this prefix. It is important to note that this is an instance of a *hyperproperty* because it characterizes *sets* of executions, i.e., all possible extensions of a prefix, instead of individual executions as in standard safety or liveness properties.

To explain the usefulness of binding, we use the execution template on the right of Figure 1. This defines non-terminating executions of the consensus protocol against a specific adaptive adversary assuming a "worst-case" BCA protocol, which satisfies the specification described in Section 2.1 but does *not* satisfy binding. Therefore, assuming two processes with different inputs, for every round r, the adversary schedules BCA so that a first process returns $\perp$ and the second process's return value is not yet fixed. Then, it schedules the first process to get a value $c_r \in \{0, 1\}$ from the common coin and after observing this value, it resumes BCA so that the second process gets the value $1 - c_r$ (this is admitted by the BCA specification). The conditional at lines 6–12 implies that the first process will enter the next round with *est* being the outcome of the coin toss, and the second process with *est* being the value returned by BCA. Therefore, they enter the next round with different estimations of the agreement value, and the same can be repeated infinitely often. Since this repeats for all possible outcomes of the coin tosses, non-termination happens with probability 1.

Note that this would not be possible for both outcomes $c_r \in \{0, 1\}$ of the coin toss if BCA satisfies binding. Indeed, after the first process gets $\perp$ from BCA (and before the coin toss), the value returned by BCA to the second process is *fixed* in *any* possible extension, i.e., it is the same no matter the outcome of the coin toss. Therefore, for one of the two possible outcomes of the coin toss, this return value equals that outcome, and the two processes will enter with equal values of *est* in the next round.

When binding holds, an adaptive adversary can *not* impose the schedule described above and the protocol terminates with probability 1. In every round, if the BCA value is not $\perp$, then it equals the outcome of the coin toss with probability $1/2$, which leads to outputting a decision. If all processes get $\perp$ from BCA, then the common coin leads directly to agreement. Therefore, the protocol terminates within a constant expected number of rounds.

2.3 Preserving binding

In the context of this consensus protocol, we discuss the possibility of replacing a given BCA protocol with a block DAG simulation as defined by Schett and Danezis [17]. The results in [17] are not sufficient to deduce that the block DAG simulation satisfies binding if the original protocol did, because, as mentioned above, binding is an instance of a hyper-property and hyper-properties are not preserved by standard trace inclusion [2]. Therefore, based on the results in [17], the proof of termination that assumed binding is not applicable to the block DAG simulation.

In this work, we present a block DAG simulation that handles protocols that use public-coins and shared objects (including a common coin like Toss). We establish that it is a *forward simulation*, which by previous work [2], implies that the set of traces defined by an adaptive adversary of the consensus protocol with the original BCA protocol is the same when the latter is replaced with the block DAG simulation (the results in [2] were applied in the context of concurrent objects and programs using such objects, but they are stated in terms of LTSs models of such programs and apply more generally to distributed protocols as well). Therefore, if one satisfies binding, then the other one satisfies it as well. This is enough to conclude that the termination argument used for the original protocol holds for the block DAG simulation as well.

3 Preliminaries

For any $n \in \mathbb{N}$, we denote $[n] = \{1, \ldots, n\}$. For any two strings s_1 and s_2, we denote by $s_1 \circ s_2$ the concatenation of the two strings.

We consider an asynchronous network with n processes $p_1, \ldots, p_n$. Each process p_i has a local process state PS_i, and buffers $In_{j\to i}$ and $Out_{i\to j}$, for each $j \in [n]$, that serve for communicating with p_j, as well as a buffer $Rqsts_i$ that contains incoming user requests. A schedule consists of two types of events:

- A $\mathsf{compute}(i)$ event lets process p_i receive *all* the messages in the buffers $In_{j\to i}$, as well as the requests in $Rqsts_i$, and update the local state PS_i. The local computation performed to update PS_i may result in new messages being deposited in the outgoing buffers $Out_{i\to j}$ and indications being sent to the user.
- A $\mathsf{deliver}(i, j)$ event moves the *oldest* message in $Out_{i\to j}$ to $In_{j\to i}$.

We assume a computationally bounded adversary that may adaptively corrupt up to f processes, and also controls the scheduling of the system. Initially, all n processes are *correct* and honestly follow the protocol. Once a process is corrupted, it may behave arbitrarily. The adversary can also read all messages in the system, even those sent by correct processes. Although the scheduling of message delivery is adversarial, we assume eventual delivery, i.e., every message sent is eventually delivered.

In a randomized protocol, the local computation of a process can depend on the result of local coin flips. To model this, we assume each process p_i has access to a random *tape*, from which it can draw a random string at each $\mathsf{compute}(i)$ event. Our simulation can be applied to *public-coin* protocols, which are randomized protocols that do no require processes to keep secrets, i.e., they can broadcast the random string they draw as soon as they use it. This definition captures protocols in the full-information model such as [12].

To allow for easy composition, we define *shared objects*. A shared object is an implementation of an interface that is accessible by all processes. For example, in the context of the randomized consensus protocol in Fig. 1 we used a shared object called *common coin* with a method Toss. For any shared object o, each process p_i can invoke o as it performs any local computation. Invocations are non-blocking, and o may at any point return a value in a designated buffer $\mathsf{o}.\mathit{buff}_i$. Whenever a $\mathsf{compute}(i)$ event is scheduled, the contents of $\mathsf{o}.\mathit{buff}_i$ are dequeued and may affect the local computation.

4 Modeling protocols with Labeled Transition Systems

We model a protocol as a *Labeled Transition System* (*LTS*), which is a tuple $L = (Q, \Sigma, q_{start}, \delta)$ where:

1. Q is a (possibly infinite) set of states.
2. Σ is a set of (transition) labels.
3. q_{start} is the starting state.
4. $\delta \subseteq Q \times \Sigma \times Q$ is a (possibly infinite) set of transitions, written as $q_1 \xrightarrow{l} q_2$ for any $(q_1, l, q_2) \in Q \times \Sigma \times Q$.

An execution of L is an alternating sequence of states and transition labels $\alpha = q_0, l_0, q_1, l_1, \ldots$ s.t. $q_i \xrightarrow{l_i} q_{i+1}$ for any $i \geq 0$. If there exists any partial execution $q_i, l_i, \ldots, l_{j-1}, q_j$ then we write $q_i \xrightarrow{l_i,\ldots,l_{j-1}} q_j$. We define a subset of labels $\Sigma_E \subseteq \Sigma$ as the *external actions*, and define a *trace* of L to be the projection of an execution over Σ_E. Typically, external actions correspond to requests and indications in the interface of a protocol, and define the "observable" behavior of a protocol. For instance, the external actions of a consensus protocol are about setting the input of each process and outputting their decisions.

LTSs as defined above can be used to model deterministic protocols in a straightforward manner. Essentially, LTS states correspond to tuples of states of participating processes and communication channels, and each transition corresponds to a step of some process (more details are given below).

Randomized protocols can be modeled using an extension of LTSs called *(simple) probabilistic automata* [18] where a transition from a state q leads to a probability distribution over states instead of a single state. The semantics of a probabilistic automaton is formalized in terms of *probabilistic executions*, which are probability distributions over executions defined by a deterministic scheduler that resolves the non-determinism. *Probabilistic traces* are defined as projections of probabilistic executions to external actions (similarly to the non-probabilistic case). The deterministic scheduler corresponds to the notion of adaptive adversary described above which controls message delivery and process scheduling. To simplify the formalization, we model randomized protocols using LTSs instead of probabilistic automata by including results of random choices in the transition labels. The transition labels corresponding to random choices are defined as external actions. The relevance of this modeling choice will be detailed later when discussing forward simulations.

Let $\mathcal{P}$ be a public-coin protocol and $\mathcal{O}$ be a *set* of shared objects used by $\mathcal{P}$. We define the LTS of $\mathcal{P}$ as follows $L = (Q, \Sigma, q_{start}, \delta)$. A state $q \in Q$ consists of the local state PS_i, the incoming messages $(In_{j\to i})_{j\in[n]}$, the outgoing messages $(Out_{i\to j})_{j\in[n]}$ and the incoming object return values $(\mathsf{o}.\mathit{buff}_i)_{\mathsf{o}\in\mathcal{O}}$ of each process p_i. For convenience, we assume that incoming user requests are stored in $In_{i\to i}$ and outgoing user indications are stored in $Out_{i\to i}$. Overall, $q = \big(PS_i, (In_{j\to i})_{j\in[n]}, (Out_{i\to j})_{j\in[n]}, (\mathsf{o}.\mathit{buff}_i)_{\mathsf{o}\in\mathcal{O}}\big)_{i\in[n]}$. We use register notation to refer to the components of each state, e.g., $q.In_{j\to i}$ refers to the incoming messages buffer from j to i in the state q. In the initial state q_{start}, all of the processes have the initial local state and all of the message buffers are empty. For the consensus protocol in Fig. 1, local states are valuations of r, val, c, and est, and the buffer for incoming object return values will contain values returned by Toss. User indications are decision values outputted at line 7.

The transition labels Σ correspond to the different types of steps in a protocol execution, namely, local computation, message delivery, return values from objects in $\mathcal{O}$, or user requests and indications. Observe that we do not need to label sending requests to $\mathsf{o} \in \mathcal{O}$ as this is done in an ordinary local computation event. In addition, the local computation label would include the randomness (if any) that is used by the process in the said computation event. Formally, the labels in Σ are as follows:

1. $\mathsf{compute}(i,\rho)$ denotes a transition where process p_i performs a local computation with ρ as its randomness. For the consensus protocol in Fig. 1, a local computation step would consist in assigning a value to est depending on the conditions starting with line 6.
2. $\mathsf{deliver}(i \to j)$ denotes a transition where all messages in $Out_{i\to j}$ are moved to $In_{i\to j}$.
3. $\mathsf{o.indicate}(i,w)$ denotes a transition where the value w has been added to $\mathsf{o}.\mathit{buff}_i$. In Fig. 1, this would correspond to the common coin object returning a value for Toss.
4. $\mathsf{request}(i,x)$ denotes a transition where process p_i receives x as input. In Fig. 1, this models a process receiving an input value to use in the consensus protocol.
5. $\mathsf{indicate}(i,y)$ denotes a transition where process p_i returns y as output. In Fig. 1, this corresponds to the output at line 7.

The external actions in $\Sigma_E \subseteq \Sigma$ are user requests ($\mathsf{request}(i,x)$) and indications ($\mathsf{indicate}(i,y)$), and local computation events ($\mathsf{compute}(i,\rho)$). The latter are included in Σ_E in order to be able to relate probability distributions in different protocols, as discussed hereafter. A transition $(q_1,l,q_2) \in Q\times\Sigma\times Q$ is in δ if and only if the protocol can get from state q_1 to state q_2 by executing the step denoted by the label l.

5 Forward simulations

Showing that a block DAG protocol is a "correct" simulation of some other protocol relies on the notion of *forward simulation* between the LTSs modeling the two protocols, respectively.

▶ **Definition 1** (forward simulation). *Let $L = (Q,\Sigma,q_{start},\delta)$ and $L' = (Q',\Sigma',q'_{start},\delta')$ be two LTSs with the same set of external actions Σ_E. A relation $R \subseteq Q\times Q'$ is a* forward simulation *from L to L' if both of the following hold:*

- $(q_{start}, q'_{start}) \in R$
- *For any $(q_1,l,q_2)\in\delta$ and any q'_1 such that $(q_1,q'_1)\in R$, there exists $q'_2\in Q'$ such that:*
  - $(q_2,q'_2)\in R$,
  - *$q'_1 \xrightarrow{\sigma} q'_2$ is a partial execution of L' (σ is a sequence of labels in Σ'), and*
  - *if $l\in\Sigma_E$, then the projection of the label sequence σ over Σ_E is exactly l.*

When L is an LTS modeling a block DAG simulation of a deterministic protocol $\mathcal{P}$ that is modeled as an LTS L', the existence of a forward simulation R from L to L' implies that the set of traces of L is included in the set of traces of L' [15]. It also implies the preservation of (hyper-)properties of *finite* probabilistic traces of randomized protocols when some sub-protocol $\mathcal{P}$ is replaced by a block DAG simulation of it [2] (a concrete example was given in Section 2). If the forward simulation is *weak progressive* [8], i.e., there exists a well-founded order such that if $\sigma = \epsilon$ in Definition 1 then either q_2 is smaller than q_1 in this order or there exists an infinite execution from q_2' with empty trace, then (hyper-)properties of *infinite* probabilistic traces are also preserved.

These results extend to randomized protocols as well. Assuming that the random choices follow the uniform distribution, a forward simulation would imply that any random choice in L is mimicked in precisely the same manner by L'. This is because the label of every step that includes a random choice is an external action and the result of that random choice is included in the label itself. This holds even for *non-uniform* random sampling as long as probabilities are recorded in transition labels. More formally, it will imply the existence of a *weak probabilistic simulation* which is known to imply that the probability distributions over traces of L defined by a deterministic scheduler are included in the probability distributions over traces of L' defined by a deterministic scheduler [18]. Moreover, it will also imply the preservation of probability distributions over executions of programs that use the block DAG simulation instead of the original protocol (this is a consequence of weak probabilistic simulations being sound for the trace distribution precongruence [18]).

Therefore any standard specification of a protocol, e.g., safety or (almost-sure) termination against an adaptive adversary, is preserved by a block DAG simulation provided the existence of a forward simulation. Moreover, typical specifications of programs using the DAG simulation instead of the original protocol will also be preserved.

6 Block DAGs

A *block* is the main type of message that is exchanged in DAG-based protocols and our block DAG simulations. A block issued by some process p_i allows p_i to: (1) inject data into the system, e.g., user inputs or shared object outputs, and (2) establish a dependency between events of different processes. To that end, the main fields of a block B are the identity of the issuing process $B.p$, injected data $B.d$, and references to other blocks $B.preds$ (on which B directly depends). The reference of B is denoted by $\mathsf{ref}(B)$.

We require that each reference must uniquely identify a specific block. One way to achieve this is using *cryptographic collision resistant hash functions*: the reference $\mathsf{ref}(B)$ consists of a hash of the block B. By the collision resistance of the hash function, it is infeasible for a computationally bounded adversary (or correct processes) to issue two distinct blocks that hash to the same value and this ensures that the reference identifies a unique block.

Since blocks are supposed to represent local computation, and local computation steps of any one process are always totally ordered, then each block B must include one reference to a parent block which we denote by $B.parent$, except for one *genesis* block for each process which does not have a parent. In addition, all of the blocks issued by one honest process should form a chain, i.e., a directed path that starts with the genesis block.

We define the *ancestors* of a block B to be all of the predecessors of B, and their predecessors and so on; this set is denoted $\mathsf{ancestors}(B)$.

A block B is *authentic* if it was issued by the process $B.p$. It is crucial to ensure the authenticity of each block before allowing it into the system. Otherwise, faulty processes can impersonate honest processes and sabotage safety properties. We can ensure authenticity by using a *cryptographic digital signature scheme*. That is each process must sign each block it issues, and other processes validate the block by checking the signature attached to it.

Ensuring that each individual block is authentic is not enough to ensure that only authentic blocks enter the system. We should also require that a block depends only on authentic blocks, that is $\mathsf{ancestors}(B)$ must all be authentic in order for B to enter. We say that a block is *valid* if it is authentic and all of $B.preds$ are valid. Note that this recursive definition is equivalent to requiring $\mathsf{ancestors}(B)$ all be authentic. Following this discussion, to ensure safety, only valid blocks would be considered by correct processes. When a process p_i validates a block B, we write $\mathsf{valid}(p_i, B)$.

Each process p_i maintains a local DAG G_i consisting of the valid blocks that p_i receives as nodes and includes a directed edge $B' \to B$ if and only if $B' \in B.preds$. Note that we need a mechanism for p_i to ensure that G_i is a DAG. A simple mechanism would be for p_i to validate B only after it has validated $B.preds$ and not validate multiple blocks "atomically". This alongside the fact that each reference identifies a unique block, would ensure that no block in a directed cycle would ever be considered valid. Formally, a *Block DAG of a correct process* p_i is a graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ such that

- $V_\mathcal{G} \subseteq \{B : \mathsf{valid}(p_i, B)\}$.
- If $B \in V_\mathcal{G}$ then for all $B' \in B.preds$ it holds that $B' \in V_\mathcal{G}$.
- $E_\mathcal{G} = \{(B', B) \in V_\mathcal{G} \times V_\mathcal{G} : B' \in B.preds\}$.
- $\mathcal{G}$ is acyclic.

Observe that by the definition of $\mathcal{G}$, for every $B \in V_\mathcal{G}$ it holds that $\mathsf{ancestors}(B) \subseteq V_\mathcal{G}$. When $B' \in \mathsf{ancestors}(B)$, we write $\mathsf{path}(B', B)$.

7 Simulating Public-Coin Protocols That Use Shared Objects

Simulating a protocol on a block DAG consists of two components: first, a mechanism that allows processes to build and maintain a *joint block DAG* and second, an algorithm to *interpret* this joint block DAG as an execution of the original protocol. Given those two ingredients, we can execute an instance of the protocol without sending any actual messages that are specific to the protocol itself. Of course, maintaining the joint block DAG would require exchanging one type of message (block), but those messages are agnostic to the protocol being simulated. This means that we can use the same joint block DAG to interpret multiple instances of the same protocol or even instances of different protocols.

Figure 2 describes how to simulate a public-coin protocol $\mathcal{P}$ using the components mentioned above. We refer to this protocol as the *block DAG simulation of* $\mathcal{P}$ and denote it by $\mathsf{BD}(\mathcal{P})$. We allow $\mathsf{BD}(\mathcal{P})$ to access the same shared objects as $\mathcal{P}$.

Simulation of Public-Coin Protocols on Block DAGs

From the perspective of process p_i, user requests go directly to $Rqsts_i$.
Initialize $G_i = (V_i, E_i)$ with $V_i = \{B_j\}_{j\in[n]}$ where B_j is a dummy genesis block for the process p_j. On every $\mathsf{compute}(i)$ event:

1. Run $\mathsf{genBlock}(G_i, blks)$.
2. If new blocks were added to G_i, then run $\mathsf{interpret}(G_i, \mathcal{P})$.
3. Run $\mathsf{exchangeBlocks}(G_i, blks)$.

Figure 2 The simulation algorithm for public-coin protocols

Interpreting the block DAG as an execution of $\mathcal{P}$ is done using the $\mathsf{interpret}$ algorithm, described in Section 7.1. This algorithm runs locally and involves no communication, yet guarantees that if two correct processes are interpreting the same (partial) block DAG, then their interpretations would be identical.

Maintaining the joint block DAG is done using the genBlock and exchangeBlocks algorithms (discussed in Section 7.2): genBlock is responsible for creating new blocks and exchangeBlocks is responsible for passing those blocks around to ensure that all correct processes receive the same blocks even if the process that issued the block is corrupted.

The aforementioned components, together, ensure that correct processes have consistent views of the execution of $\mathcal{P}$ at all times. However, this does not guarantee that the execution is useful, e.g., it might give the adversary more power or it might be a “liveless” execution where the correct processes are not making any progress. For that reason, we prove in Section 7.3 that the execution (defined by the views) is faithful in the sense that there exists a forward simulation towards the original protocol. This guarantees that the simulation of $\mathcal{P}$ on the block DAG preserves $\mathcal{P}$'s original specification.

7.1 Common Interpretation

Given a block DAG $\mathcal{G} = (V, E)$, we want to interpret it as an execution of the protocol. We call this execution the *simulated execution.* Furthermore, we need the interpretation to be consistent among all correct processes doing it.

The idea is to view $\mathcal{G}$ as a causality graph, where a block in $\mathcal{G}$ issued by some process p_i corresponds to a node that belongs to p_i in the causality graph, and the node corresponds to a compute(i) in the simulated execution. In order to interpret $\mathcal{G}$, we interpret each block separately, where the interpretation of the block consists of the local process state and its outgoing messages after the corresponding compute(i) event. For convenience, we also treat the incoming messages (right before the event) as part of the interpretation. Formally:

▶ **Definition 2** (Block Interpretation). *The* interpretation *of a block B has the following fields:*
1. *A local process state $B.PS$.*
2. *A list of incoming messages $B.M_{in}$.*
3. *A list of outgoing messages $B.M_{out}$. For convenience, we denote by $M_{out}[j]$ the outgoing messages in M_{out} that are designated to p_j.*

Note that the interpretation of a block is *not* sent over the network. This is crucial because we do not want the size of the block sent over the network to increase with the number of protocol instances being interpreted, and instead we only want the block to include information that processes cannot locally compute unambiguously. As such, it is the responsibility of each process to interpret each block it has locally.

In a regular execution of a *deterministic* protocol, whenever a compute(i) event is scheduled, the process p_i performs the following: it passes all of the message in $In_{j\to i}$ to the local state of its protocol instance PS_i and performs a local computation. This updates the local state PS_i, produces new outgoing messages that are deposited into $Out_{i\to j}$ and may return user indications. Our interpretation protocol tries to mimic the execution by assigning to $B.PS$ the local state of the process after the corresponding event, $B.M_{out}[j]$ the messages that would be deposited in $Out_{i\to j}$, and $B.M_{in}$ the messages that would have been in $In_{j\to i}$ before the event. In addition, if the block B was issued by the process doing the interpretation and $B.PS$ produces a user indication, then the process must actually return the indication to the user. The way to compute $B.PS$ is as follows: $B.PS$ is initially copied from the parent block (or initialized as an initial state for genesis blocks), and then we feed it all of the relevant outgoing messages from the interpretation of the predecessor blocks, that is all messages in $B'.M_{out}[i]$ for all $B' \in B.preds$, where $B.p = p_i$.

Algorithm 2 interpret$(G_i, \mathcal{P})$ for process p_i

$G_i = (V_i, E_i)$ is a block DAG and $\mathcal{P}$ is a public-coin protocol.
G_i is process-local variable that maintains its value across different invocations

while $\exists B \in G_i$ s.t. B is not interpreted s.t. $\forall B' \in B.preds : B'$ is interpreted **do**
  if $B.k = 0$ **then**
    Initialize $B.PS$ as a new state according to the protocol $\mathcal{P}$ and process $B.p$
  else
    $B.PS := B.parent.PS$
  for all $B' \in B.preds$ **do**
    Copy messages from $B'.M_{out}[B.p]$ to $B.M_{in}$
  Pass the user requests $B.rqsts$, messages $B.M_{in}$, random tape $B.rand$ and the object indications $B.buff$ to the state $B.PS$
  Overwrite the new state in $B.PS$
  Store the outgoing messages in $B.M_{out}$
  if $B.p = i$ **then**
    Return user indications produced by $B.PS$ to the user
    Perform object invocations as dictated by $B.PS$

When extending this approach to *randomized* protocols, we need to account for the local randomness. In this case, the process state expects to additionally receive a random tape. It is the responsibility of the issuing process to include the tape in the block B and attach it as a part of the block in a data field $B.rand$. The interpretation is thus similar to that of a deterministic protocol, but $B.rand$ is now also passed to the process state as randomness.

When further extending this to protocols with *shared objects*, we need to handle object invocations and object indications. In a regular execution of a protocol with a shared objects o, a process p_i might invoke o following a compute(i) event. Similarly, when interpreting a block, $B.PS$ might dictate that $B.p$ should invoke o. In this case, the interpreting process p_i actually performs the invocation only if it is the issuing process of the block $p_i = B.p$. The process states in the original protocol expect to receive indications from o, so these indications should be passed to $B.PS$ when interpreting B. When o returns an indication to p_i, it is the responsibility of p_i to attach the indications to the block in a special buffer $B.buff[\mathsf{o}]$. The contents of $B.buff[\mathsf{o}]$ are passed to $B.PS$ when interpreting B. This concludes the high level description of block interpretation. In order to interpret an entire block DAG, we interpret blocks in a topological order since the interpretation of each block B depends on the interpretation of its predecessors. Since the graph is a DAG, such an order exists and every block can be interpreted. The full algorithm interpret$(\mathcal{G}, \mathcal{P})$ is presented in Algorithm 2. The main guarantee of interpret$(\mathcal{G}, \mathcal{P})$ is the fact that the interpretation of B is independent of $\mathcal{G}$. This is formalized in the following lemma (proved in the full version [3]):

▶ **Lemma 3.** *For any two block DAGs G_1 and G_2, if $B \in G_1$ and $B \in G_2$ then the interpretation of B in both* interpret$(G_1, \mathcal{P})$ *and* interpret$(G_2, \mathcal{P})$ *is identical.*

7.2 Joint Block DAG

In this section, we demonstrate how processes build and maintain the block DAGs that they interpret in Section 7.1. Algorithm 3 presents the genBlock(G_i) algorithm, which allows a process to generate blocks and inject data into the system.

Algorithm 3 $\mathsf{genBlock}(G_i)$ for process p_i

$G_i = (V_i, E_i)$ is a block DAG.

1: Initialize a new block B as follows $B.p := p_i, B.preds := \emptyset, B.rqsts := \emptyset$
2: Assign to $B.parent$ the reference of the most recent block in G_i issued by p_i.
3: $B.k := B.parent.k + 1$
4: **for all** $B' \in V_i$ s.t. $\neg\mathsf{path}(B', B.parent)$ **do**
5: $\quad B.preds := B.preds \cup \{\mathsf{ref}(B')\}$
6: Fill the external data $\mathsf{fillData}(B)$.
7: **return** B

The algorithm gets a block DAG G_i which is assumed to be a valid block DAG of p_i. It then generates a new block B and assigns it a parent from G_i, then adds to $B.preds$ all references to blocks in G_i that do not have a path to $B.parent$. Note that since $B.preds \subseteq V_i$, then $B.pred$ only includes blocks B' s.t. $valid(p_i, B')$. This guarantees that B is a valid block. Next the external data is filled into the block: this includes moving the user requests from $Rqsts_i$ to $B.rqsts$, moving the object indications from $\mathsf{o}.buff_i$ to $B.buff[\mathsf{o}]$ for each relevant $\mathsf{o} \in \mathcal{O}$ and finally assigning a random string ρ to $B.rand$. Note that we do not know exactly how long ρ needs to be until B is actually interpreted. Since all $B' \in B.preds$ are already in G_i, process p_i can already interpret B and generate ρ while generating B.

Next, we describe the communication component that is responsible for exchanging blocks and growing the DAGs. We have shown that processes that interpret the same blocks reach the same conclusion. But for this to be useful, the communication component must ensure correct processes eventually interpret the same blocks. That is, if a correct process p_i adds some B to G_i, then every correct process p_j eventually adds B to G_j. This can be viewed as a consistency property between two processes.

Note that a naive approach of having each process simply send its blocks to everyone does not guarantee consistency, since an honest process p_i may add a block B^* by some corrupted process B^* as a predecessor for its own block B. p_i naturally considers B valid and adds it to its block DAG, but for any other honest process p_j, B will never be considered valid until it receives B^* from p^*.

Consistency can be achieved using a simple *echoing* mechanism that we describe now. For each block B that p_i issues using $\mathsf{genBlock}$, p_i generates a signature for B which we denote by $B.\sigma$, and sends $(B, B.\sigma)$ to everyone. When p_i receives a block B by some other process, it first ensures B is authentic (by verifying the signature). After collecting all authentic blocks, p_i tries to validate as many of them as possible. The validation may only fail if some $B' \in B.preds$ of B is missing, so p_i requests B' from the process $B.p$ that issued B, using a forward request message which we denote by $\mathsf{FWD}(\mathsf{ref}(B'))$. The idea is that if $B.p$ is correct then it must have those blocks, so it will eventually send them to p_i, allowing p_i to validate the block $B.p$. Finally, p_i of course has to respond to the forward requests it has received. This concludes the informal description of $\mathsf{exchangeBlocks}$. The consistency guarantee is formalized in the following lemma:

▶ **Lemma 4.** *For any two correct processes p_i and p_j executing the protocol of Figure 2, if p_i adds a block to its block DAG G_i, then p_j eventually inserts B into G_j.*

We note that Lemma 4 really refer to any protocol in which Algorithms 3 and 4 are continuously run, and are not specific to Figure 2. The proof is deferred to the full version [3].

Algorithm 4 $\mathsf{exchangeBlocks}(G_i)$ for process p_i

$G_i = (V_i, E_i)$ is a block DAG
toValidate and *isSent* are process-local variables that maintain their values across different invocations
Initialize $toValidate := \emptyset$ and $isSent := \emptyset$

for all $B \in G_i$ s.t. $B.p = p_i$ and $B \notin isSent$ **do**
    Sign B and denote the signature by $B.\sigma$
    Send $(B, B.\sigma)$ to everyone
Move all authentic blocks from all $In_{j\to i}$ to a set *auth*
$toValidate := toValidate \cup auth$ ▷ Throw inauthentic blocks
while $\exists B \in toValidate$ s.t. $\mathsf{valid}(p_i, B)$ **do**
    $G_i.\mathsf{insert}(B)$
    $toValidate := toValidate \setminus \{B\}$
    $auth := auth \setminus \{B\}$
for all $B \in auth$ **do** ▷ Try to validate all authentic blocks
    for all $B' \in B.preds$ s.t. $B' \notin G_i$ **do**
        Send $\mathsf{FWD}(\mathsf{ref}(B'))$ to $B.p$ ▷ Request missing blocks from $B.p$
for all $\mathsf{FWD}(\mathsf{ref}(B'))$ in some $In_{j\to i}$ **do** ▷ Respond to forward requests
    If $B' \in G_i$, send $(B', B'.\sigma)$ to p_j
Empty all $In_{j\to i}$.

7.3 Correctness Proof

Combining Lemma 4 with Lemma 3 and assuming eventual delivery of blocks, we get eventual delivery of simulated messages. In other words, if a correct process p_i wants to send a message m to some correct process p_j, then this is expressed in the block DAG framework as a block B issued by p_i, such that $B.M_{out}[j]$ contains the message m. Delivering the message m to p_j is expressed by p_j creating a block B' such that $m \in B'.M_{in}$. Note that referring to unambiguous interpretations of B and B' is only possible through Lemma 3. By Lemma 4, we know that if p_i issues the block B then p_j eventually receives B and considers it valid. By the algorithm in Algorithm 3, eventually p_j creates a new block B' such that $B \in B'.preds$ and by Algorithm 2, m will be added to $B.M_{in}$. This discussion demonstrates that the block DAG framework guarantees eventual delivery of simulated messages, if we assume eventual delivery of blocks. This guarantees the liveness of the block DAG simulation.

We show that the block DAG simulation of a protocol $\mathcal{P}$ is faithful in the sense that there exists a *forward simulation* from the block DAG simulation denoted as $\mathsf{BD}(\mathcal{P})$ to $\mathcal{P}$ (modeled as LTSs). As mentioned in Section 5, this implies that the block DAG simulation inherits finite-trace probability distributions of $\mathcal{P}$ and that typical specifications of programs using the DAG simulation instead of $\mathcal{P}$ are preserved.

Section 3 describes the modeling of $\mathcal{P}$ using LTSs. We describe below a modeling of $\mathsf{BD}(\mathcal{P})$ using an LTS $L' = (Q', \Sigma', q'_{start}, \delta')$ which simplifies the forward simulation proof. A state $q' \in Q'$ contains the block DAG G_i of each process p_i and $(In^B_{j\to i})_{j\in[n]}$ and $(Out^B_{i\to j})_{j\in[n]}$ for each process p_i, where $In^B_{j\to i}$ is the incoming buffer of process i with blocks sent by process j and $Out^B_{i\to j}$ is the outgoing buffer with blocks sent by i to j. As before, we assume that incoming user requests are stored in $In^B_{i\to i}$ and outgoing user indications are stored in $Out^B_{i\to i}$. The shared object indications are stored in separate buffers $(\mathsf{o}.buff_i)_{\mathsf{o}\in\mathcal{O}}$ as before.

Overall, $q' = \big(G_i, (In^B_{j\to i})_{j\in[n]}, (Out^B_{i\to j})_{j\in[n]} (\mathsf{o}.\mathit{buff}_i)_{\mathsf{o}\in\mathcal{O}}\big)_{i\in[n]}$. In the initial state q'_{start}, all of the block DAGs and the buffers are empty. The transition labels correspond to computing and validating blocks, exchanging blocks, and user requests or indications. In comparison to the "standard" model described in Section 3 we decompose a compute step of a process as defined in Figure 2 into a sequence of steps. This simplifies the forward simulation proof. As before, we include the randomness (that is attached to the newly created block) in the computation label. Formally, the transition labels are as follows:

1. $\mathsf{validateBlock}(i \to j)$ denotes a transition where p_j validates a block issued by p_i (inside the $\mathsf{genBlock}$ algorithm).
2. $\mathsf{compute}(i, \rho)$ denotes a transition where process p_i produces and disseminates a new block (inside the $\mathsf{genBlock}$ algorithm) with ρ as its randomness, and then runs $\mathsf{interpret}$ to interpret the new block (and other previously uninterpreted blocks).
3. $\mathsf{sendFWD}(i \to j)$ denotes a transition where p_i sends a FWD request to p_j.
4. $\mathsf{replyFWD}(i \to j)$ denotes a transition where p_i sends a reply to a FWD sent by p_j.
5. $\mathsf{deliverBlocks}(i \to j)$ is a transition where all the blocks in $Out^B_{i\to j}$ are moved to $In^B_{i\to j}$.
6. $\mathsf{o.indicate}(i, w)$ denotes a transition where the value w has been added to $\mathsf{o}.\mathit{buff}_i$.
7. labels for user requests ($\mathsf{request}(i, x)$) or indications ($\mathsf{indicate}(i, y)$) are used as in Section 3.

The external actions Σ_E are defined exactly as for the LTS L modeling $\mathcal{P}$, presented in Section 3 (Σ_E includes $\mathsf{request}(i, x)$, $\mathsf{indicate}(i, y)$, and $\mathsf{compute}(i, \rho)$). A transition $(q'_1, e, q'_2) \in Q' \times \Sigma' \times Q'$ (denoted $q'_1 \xrightarrow{e} q'_2$) is in δ' if and only if the protocol $\mathsf{BD}(\mathcal{P})$ can get from state q'_1 to state q'_2 by executing the step denoted by the label e. Theorem 5 is proved in the full version [3].

▶ **Theorem 5.** *There exists a forward simulation from the LTS L' modeling* $\mathsf{BD}(\mathcal{P})$ *to the LTS L modeling $\mathcal{P}$.*

8 Relation to Prior Work

Comparison with the deterministic simulation. We can now discuss how our simulation and proof are related to the work of Schett and Danezis [17]. They show how block DAGs can be used to simulate deterministic protocols, which are a special case of the protocols that we handle here. Readers that are familiar with their work will notice that we were able to achieve a simulation that is a natural extension of theirs. We emphasize, however, that our techniques for proving the faithfulness of our simulation are novel and different from theirs. This is necessary because their techniques do not capture the probabilistic guarantees of randomized protocols.

Our network component which consists of $\mathsf{genBlock}$ and $\mathsf{exchangeBlocks}$ algorithms is a natural extension of the gossip algorithm of [17]. Indeed, the code responsible for generating new blocks and echoing them is almost identical to that of gossip. The difference is that because we want to exchange only blocks, they should carry enough information to resolve the randomized decisions that can come from local randomness or shared objects. In our protocol, each process is responsible to pass along its local randomness or the indications it got from the shared object in the blocks that it creates. Lemma 4 is proved in a manner similar to [17, Lemma 3.7].

Our interpretation algorithm is the natural extension of $\mathsf{interpret}$ algorithm of [17] for our context. That is, when interpreting a deterministic protocol, the computation of each process is only determined by the incoming messages and its state prior to processing those messages. When interpreting a randomized protocol with shared objects, the local computation may depend on local randomness and object indications. Our interpretation algorithm used those fields that were already attached to each block by our genBlock. Lemma 3 that states the common interpretation of block DAGs, is analogous to [17, Lemma 4.2]. However, the proof of the latter had a minor mistake and our proof is slightly different.

Finally, the guarantees of randomized protocols, unlike those of deterministic protocols, cannot always be expressed as trace properties. Particularly, for our simulation to be faithful to the original protocol, we need a more careful and precise statement and proof. Therefore, the modeling in Sections 3 and 7.3 as well as the proof of Theorem 5 are totally different from what appears in [17].

Analyzing existing protocols. Several recent works rely on the block DAG approach, e.g., Aleph [10], DAG-Rider [13] and Bullshark [20]. All of these protocols are randomized. While each of these works presents a new protocol, we provide a formal and systematic framework for analyzing DAG-based protocols, especially *randomized* block DAG protocols.

Here we discuss how our simulation applies to existing protocols, concentrating on Aleph [10] and DAG-Rider [13]. These protocols aim to order the blocks of the DAG, so as to implement *Byzantine Atomic Broadcast* (BAB). A BAB protocol allows all processes to receive the same messages in the *same order*. One natural way of implementing a BAB protocol using a block DAG is by having each process attach the messages it wants to broadcast to a block and then broadcast the block to everyone. The processes then just need to agree on an order of the blocks, which would induce an order of the messages.

Analogous to our simulation, both Aleph and DAG-Rider have a communication component that is responsible for building and maintaining the common DAG. In both protocols, each block in the DAG belongs to a specific round, and each correct process has a single block in each round.

Aleph orders the blocks in the DAG by electing a leader block in each round, and then having that leader block (deterministically) dictate the order of its ancestor blocks that have not been ordered yet.

DAG-Rider divides the DAG into *waves*. Each wave consists of four consecutive rounds, and a leader block is elected for each wave. The block leader election is done by interpreting the (same) block DAG as a consensus protocol and utilizing a shared object for generating randomness, namely, a common coin. It is critical to note that our simulation preserves the properties of the shared object, for example the *unpredictability* of the common coin. This is because our forward simulation preserves the compute events, in which the object invocations happen. This means that the object cannot distinguish if it is being used in the context of the original protocol or in the context of the block DAG simulation of the protocol. This means that its properties are preserved.

Aleph and DAG-Rider can be analyzed using our framework. The consensus protocol used can be analyzed independently of Aleph or DAG-Rider, while assuming it has access to a common coin. By Theorem 5, the simulation of the consensus protocol on the block DAG is faithful to the original consensus protocol. This not only simplifies reasoning about safety and liveness of Aleph and DAG-Rider, but also supports *modularity*: the simulated consensus protocol in Aleph or DAG-Rider can be seamlessly replaced using Theorem 5.

9 Discussion

We have presented a faithful simulation of DAG-based BFT protocols, which use public coins and shared objects, including protocols that utilize a common source of randomness, e.g., a *common coin*. Being faithful, the simulation precisely preserves properties of the original

BFT protocol, and in particular, their probability distributions.

One of the appealing properties of our block DAG framework is that it allows to minimize the communication when running multiple instances of potentially different protocols. This can be done by using the same joint block DAG to interpret multiple protocol instances. The logic of the communication layer does not change, other than the need to specify the associated instance for each user request and object indication that is attached to the blocks. Each process would then run multiple interpretation instances, one for each protocol instance. We note that a process does not necessarily need to attach a separate randomness tape for each instance, and can instead attach a small random seed. Processes can then use a *pseudorandom generator* to expand the seed to a large enough pseudorandom string that can be used for all of the instances. This ensures that block size does not grow beyond the size of the user requests and the object indications.

Our simulation relies on the fact that it is safe to reveal the randomness to the adversary as soon as it is used. We can similarly define *private-coin* protocols, whose security relies on processes ability to keep secrets from the adversary. A classical example would be any Asynchronous Verifiable Secret Sharing scheme (e.g. [5]). From a theoretical point of view, it would be interesting to demonstrate how we can simulate such algorithms on block DAGs. However, we note that some protocols are entirely public-coin other than a dedicated private-coin sub-protocol, such as Aleph-Beacon in Aleph [10] (which is used to implement a common coin). In this case, the dedicated sub-protocol can be encapsulated as a shared object, thus factoring out the use of private-coin simulations.

References

1. Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 381–391. ACM, 2022. `doi:10.1145/3519270.3538426`.
2. Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programsthat use concurrent objects. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.DISC.2019.2`.
3. Hagit Attiya, Constantin Enea, and Shafik Nassar. Faithful simulation of randomized bft protocols on block dags. Cryptology ePrint Archive, Paper 2023/192, 2023. `https://eprint.iacr.org/2023/192`. URL: `https://eprint.iacr.org/2023/192`.
4. Leemon Baird. The Swirlds Hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance. `https://www.researchhub.com/paper/337/the-swirlds-hashgraph-consensus-algorithm-fair-fast-byzantine-fault-tolerance`, 2016.
5. Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 42–51. ACM, 1993. `doi:10.1145/167088.167105`.
6. Anton Churyumov. Byteball: A decentralized system for storage and transfer of value. `https://byteball.org/Byteball.pdf`, 2016.
7. Danny Dolev. The Byzantine generals strike again. *J. Algorithms*, 3(1):14–30, 1982. `doi:10.1016/0196-6774(82)90004-9`.
8. Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. Weak progressive forward simulation is necessary and sufficient for strong observational refinement. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR*

2022, September 12-16, 2022, Warsaw, Poland, volume 243 of *LIPIcs*, pages 31:1–31:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CONCUR.2022.31.

9 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. In Ronald Fagin and Philip A. Bernstein, editors, *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA*, pages 1–7. ACM, 1983. doi:10.1145/588058.588060.

10 Adam Gagol, Damian Lesniak, Damian Straszak, and Michal Swietek. Aleph: Efficient atomic broadcast in asynchronous networks with Byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019*, pages 214–228. ACM, 2019. doi:10.1145/3318041.3355467.

11 Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In Lance Fortnow and Salil P. Vadhan, editors, *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 373–382. ACM, 2011. doi:10.1145/1993636.1993687.

12 Shang-En Huang, Seth Pettie, and Leqi Zhu. Byzantine agreement in polynomial time with near-optimal resilience. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 502–514. ACM, 2022. doi:10.1145/3519935.3520015.

13 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 165–175. ACM, 2021. doi:10.1145/3465084.3467905.

14 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

15 Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995. doi:10.1006/inco.1995.1134.

16 Sean Rowan and Naïri Usher. The Flare consensus protocol: Fair, fast federated Byzantine agreement consensus. https://flareportal.com/wp-content/uploads/simple-file-list/FCP.pdf, 2019.

17 Maria Anna Schett and George Danezis. Embedding a deterministic BFT protocol in a block DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 177–186. ACM, 2021. doi:10.1145/3465084.3467930.

18 Roberto Segala. A compositional trace-based semantics for probabilistic automata. In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*, volume 962 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 1995. doi:10.1007/3-540-60218-6_17.

19 Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. PHANTOM GHOSTDAG: a scalable generalization of nakamoto consensus: September 2, 2021. In Foteini Baldimtsi and Tim Roughgarden, editors, *AFT '21: 3rd ACM Conference on Advances in Financial Technologies, Arlington, Virginia, USA, September 26 - 28, 2021*, pages 57–70. ACM, 2021. doi:10.1145/3479722.3480990.

20 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2705–2718. ACM, 2022. doi:10.1145/3548606.3559361.

21 Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. SoK: Diving into DAG-based blockchain systems. *CoRR*, abs/2012.06128, 2020. URL: https://arxiv.org/abs/2012.06128, arXiv:2012.06128.