Randomization Models in Distributed Algorithms

Shafik Nassar

July 23, 2022

1 Introduction

Deterministic algorithms behave in a predetermined way on a given input. That is, if the input is fixed, anyone can simulate the deterministic algorithm, perform the same exact computation and reach the same conclusion. Although this makes deterministic algorithms easy to analyze and reason about, it also makes them more vulnerable to *adversarial behaviour*; since the adversary knows exactly how the computation will progress, it can effectively plan an attack and compromise the liveness and/or safety of the algorithm. This vulnerability is extremely evident in distributed computing, and especially in the *asynchronous (message-passing)* model, due to the celebrated FLP result [FLP85]: consensus is impossible in the asynchronous model even if only one process crashes.

One way to overcome this impossibility result is to use *randomized algorithms* and weaken the termination condition accordingly. Randomized algorithms grant each process access to a source of random bits, which are freshly generated in each execution. This eliminates the predetermined behaviour on any given input, making randomized algorithms harder to predict and thus more secure in the presence of an adversary. Indeed, this unpredictability is sufficient to overcome the FLP barrier, as shown by a seminal line of work[1] [Rab83, Bra87, CR93, MMR15, BKLL20, ADS20]. This proves that randomization is essential to achieve *Byzantine agreement* (BA), one of the fundamental problems in distributed computing.

Categorizing Randomized Algorithms. The algorithms in [Rab83, MMR15] solve BA using a *common* source of randomness that can be securely accessed by all correct processes, this source is typically referred to as *common coin.* The algorithm in [CR93] solves BA using *local coins* without the help of a trusted party. This algorithm crucially depends on the fact that correct processes can keep their local coins a "secrets" from other (Byzantine) processes, for that reason we call these coins *private coins.* As opposed to private coins, one can think of local coins which a process generates and immediately shares with all other processes. We call those *public coins.* A more recent line [KS16, KS18, HPZ22] of work showed how to achieve BA using public-coins only.

Relative Power of Randomness Models. We can view different kinds of "coins" as different models of randomness. The aforementioned lines of work demonstrate that BA, one of the most fundamental problems in distributed computing, can be solved *efficiently* in all three models. Still, it is natural to ask about the relative power of these models. Some relationships are trivial or have already been resolved, e.g., every public-coin algorithm can be viewed as a special case of a private-coin algorithm.

[1]The list goes on and on, the authors mention only a small number of papers with which they are most familiar.

But some relationships are still wide open. For example, one question that remains open is whether we can construct a common-coin in the public-coin model. A question that we ask in this project is whether we can eliminate the need for private-coins. Namely, we ask the following question:

Question: *Is there a generic transformation from private-coin to public-coin algorithms?*

The Importance of Efficiency. When we discuss transformations between different models, or even BA algorithms, it is important to require them to be efficient. This way, we eliminate the "trivial" transformations. For example, Bracha [Bra87] shows how a very simple construction of a common-coin in the public-coin model. However, that construction has exponential expected latency and thus yields an inefficient BA algorithm (and more broadly, an inefficient transformation from common-coin to public-coin).

Indeed, it was an interesting open question whether BA can be achieved efficiently in the public-coin model[2]. Lewko showed some evidence supporting a negative result [Lew11]: protocols that do not take into account the identity of the sender nor the message history *must* have exponential expected latency. The breakthrough work of King and Saia [KS16,KS18] which was later improved in [HPZ22], gave a definite positive answer to this question by presenting a BA algorithm in *full-information*, which implies no use of private coins.

Project Purpose. In this project, we explore the different models of randomization by formally defining and surveying the seminal approaches to achieve BA in each one of them. In addition, we survey the relative power of randomness models and *present* a generic transformation from public-coin algorithms to common-coin algorithms. [3]

Organization. In Section 2, we formally define our models. In Section 3, we discuss one efficient BA algorithm for each randomness model. Finally, we discuss the connections between different models in Section 4. Namely, we present a public-coin to common-coin transformation in Section 4.2.

2 Preliminaries

2.1 Model

In this work, we consider an asynchronous network. There are n processes $\{p_i\}_{i\in[n]}$, all of which start as *correct* processes and the adversary may *adaptively* corrupt up to f processes. A corrupted process can behave arbitrarily. Each process p_i has the buffers $In_{j\to i}$ and $Out_{i\to j}$ for each $j \in [n]$ which serve for communication. An execution consists of two types of events:

- A $\mathsf{compute}(i)$ event lets p_i process all messages in the buffers $In_{j\to i}$, perform local computation (which changes its state) and deposit new messages in $Out_{i\to j}$.
- A $\mathsf{deliver}(i,j)$ event moves a message from $Out_{i\to j}$ to $In_{j\to i}$.

[2]Against an adaptive adversary. The case of non-adaptive adversary was solved earlier by [KKK+10]

[3]Admittedly, the other direction is the more interesting one, as this transformation is not practically interesting because a common-coin is much harder to achieve than a local (public) source of randomness.

We assume that the adversary can read the messages sent by correct processes. In addition, the message scheduling can be adversarial. This means that the adversary has full control over the scheduling of the message delivery, but cannot hold a message indefinitely, that is, we assume eventual delivery of messages. The adversary also cannot forge messages. The computational power of the adversary depends on the randomization model that we study.

2.2 Randomization Models

2.2.1 Common-Coin

A primitive $\mathcal{C}$ is a *common coin with ε-agreement* if the following properties hold:

1. *(Termination)* If all correct processes participate in the protocol then they all complete it. Furthermore, if some correct process completes $\mathcal{C}$, then every correct party that begins the protocol completes it.

2. *(Correctness)* For any value $b \in \{0, 1\}$, with probability at least ε, all correct processes that participate in $\mathcal{C}$ output b. The probability that two correct processes output two different values is at most $1 - 2\varepsilon$.

3. *(Unpredictability)* No Byzantine process can obtain the value of the r^{th} random bit before at least one correct process started its r^{th} invocation of $\mathcal{C}$.

If $\varepsilon = 1/2$, then we say that $\mathcal{C}$ is a *strong* common coin.

We call a protocol which uses only a common coin as its source of randomness a *common coin protocol.* When considering an execution with a common-coin $\mathcal{C}$, we treat $\mathcal{C}$ as a correct *process.* A player p_i invokes $\mathcal{C}$ by depositing a message in $Out_{i \to \mathcal{C}}$. The invocation returns when p_i receives a message from $\mathcal{C}$ (by having $\mathcal{C}$ first deposit the message, then executing a $\mathsf{compute}(i)$ event). Since the adversary cannot corrupt $\mathcal{C}$, then we treat delivering the message to $\mathcal{C}$, "local computation" of $\mathcal{C}$ and the delivery of random bit back to p_i as an atomic event $\mathsf{deliver}(\mathcal{C}, p_i)$.

2.2.2 Public-Coin

A protocol is called *public-coin* if the liveness and safety properties are guaranteed against adversaries that can access the random coins of each process as soon as they are tossed. This definition is identical to *full-information* protocols, in which the adversary can access every process's internal state, and specifically the random coins: on one hand a full-information protocol is also a public-coin protocol because the adversary can access the local coins of each correct process. On the other hand, the state of any process can be computed if given the local coins as well as the incoming messages of the process.

In public-coin protocols, for each message sent by some process, it is easy to see that the process can attach to the message the random coins used in the current round. We can view this property as if the process has *no secrets* that it wishes to hide from adversary (nor from all processes, in general). For this reason, cryptography in this model is useless, since the adversary can break the security of cryptographic primitive by looking at the random coins used by the correct processes.

In this model, it is usually assumed that the adversary is computationally unbounded.

2.2.3 Private-Coin

A protocol is called *private-coin* if the liveness and safety properties are guaranteed only against adversaries that cannot access the random coins of any correct process.

For this type of protocol, the processes rely on the fact that they can *keep secrets* from other processes, specifically, from Byzantine processes.

In this model, we usually assume that the adversary is computationally bounded and thus, we can use cryptography to encrypt messages, sign messages and commit to certain values which may be revealed later.

3 Efficient Byzantine Agreement

3.1 Common-Coin

The seminal work of Bracha [Bra87] introduces a *Byzantine reliable broadcast* primitive that can be implemented deterministically and is defined below.

Definition 1. *A Byzantine reliable broadcast algorithm is an algorithm in which there is a sender S who has a value $b \in \{0, 1\}$, such that at the end of the algorithm:*

- ***Agreement:*** *all correct processes output the same value $b' \in \{0, 1, \perp\}$.*
- ***Validity:*** *if the sender is correct then all correct processes output b.*

In addition, Bracha introduces a *message validation* mechanism which prevents corrupted processes from deviating from the protocol, that is, every message that a process sends has to be "justified" by enough messages from different processes in order to be validated. This forces the messages of corrupt players to be consistent with the protocol that we run. The main property of the validation mechanism is captured in the following lemma:

Lemma 1. *A correct process p* validates *the message of a process q in admissible execution α if and only if there exists an admissible execution β in which q is a good process and the states of all other good processes are the same in α and β.*

Leveraging reliable broadcast and message validation in a *strong* common-coin model limits the power of the adversary to malicious scheduling and dynamically corrupting correct processes. When using a weak common-coin, then the adversary may also lie about the outcome it sees from the common-coin.

Using reliable broadcast, message validation and a common-coin primitive, Bracha presents a Byzantine agreement algorithm which we write in Fig. 1.

The main takeaway from Step 1 is that if all correct processes hold the same value b, then for any set of size $n - f$, the value b will be the majority vote (because $n - f \leq 2f + 1$), therefore all correct processes will enter Step 2 with $v_i = b$. By enforcing message validation, we also guarantee that all corrupt processes have to enter Step 2 with the value b and this guarantees all processes decide in that same iteration. Another important property of Step 2 is that no two processes can have different views of the value v^*, because only one value can appear in a majority of messages that were broadcast.

When reaching Step 3, if at least one process sees a value v^* for a total of $f + 1$ times, then we know that every other process must have received v^* at least once (because it waits for $n - f$

Bracha's Byzantine Agreement

Code for process p_i. The variable v_i initially holds the input.
Repeat the following steps indefinitely:

1. Reliably broadcast v_i and wait for distinct $n-f$ processes to broadcast their *validated* value. Set v_i to be majority of the received values.
2. Reliably broadcast v_i and wait for distinct $n-f$ processes to broadcast their *validated* value. If some value v^* appears more than $n/2$ times, then set $v_i = (dec, v^*)$
3. Reliably broadcast v_i and wait for distinct $n-f$ processes to broadcast their *validated* value. Let x_i be the number of (dec, v^*) messages received.
    - If $x_i \geq 1$ then set $v_i := v^*$.
    - If $x_i \geq f+1$ then decide v^* and participate in one more iteration.
    - Otherwise, invoke $\mathcal{C}$ and set v_i to be the returned value.

Figure 1: Bracha's f-resilient Algorithm for Byzantine Agreement using a common-coin $\mathcal{C}$

values), therefore we know that all processes must enter the next round with the same value v^* and decide on it in the next iteration. We call this the *good case.*

The *common case* is when some processes see a majority value v^* (recall that we cannot have processes seeing two different as the majority) and some do not. Those that do not will invoke $\mathcal{C}$ to get a random coin. By the correctness of the common coin, with probability at least ε, all correct processes will get the value v^* and decide in the next round. Therefore, if ε is a constant then we expect the algorithm to terminate within a constant number of iterations (and thus it has constant expected latency).

This concludes our discussion of Bracha's *generic* algorithm. We sketched the proof for the agreement and validation, as well as the expected latency, in the common-coin model. Different implementations of the common-coin primitive yield different expected latency and may also transform the algorithm to a private-coin or even public-coin model.

3.2 Private-Coin

Canetti and Rabin [CR93] followed Bracha's generic algorithm and introduced a way to construct a $\frac{1}{4}$-correct common-coin in the private-coin model. Their construction is efficient and thus it proves that Byzantine agreement can be achieved efficiently in the private-coin model. The construction uses *asynchronous verifiable secret sharing* (AVSS).

describe how to construct a $\frac{1}{4}$-correct common-coin using *asynchronous verifiable secret sharing* (AVSS). In this section, we outline the construction of the common-coin and demonstrate that this result implies that a weak common-coin can be constructed using (local) private-coins.

AVSS. The construction uses *asynchronous verifiable secret sharing* (AVSS) scheme. Roughly speaking, an AVSS scheme is a pair of protocols $(\mathsf{Sh}, \mathsf{Rec})$ that allows a dealer D to share a value s (by having players invoke Sh), such that if all correct processes cooperate, they can reconstruct the value s and if no correct process has started executing Rec, then the corrupted players learn nothing about s. In addition, the scheme is verifiable in the sense that once a correct process completes Sh, then D is committed to some value s and cannot reveal anything else in Rec. So in essence, AVSS provides two functionalities: it forces corrupt players to commit to some value, and in addition it allows correct processes to secret-share their values without allowing the corrupt players to see those values (before other correct processes start reconstructing).

Common-coin from AVSS. The protocol for implementing common-coin works in two stages: in the first stage, each player p_i uses AVSS to share some values with all other players (p_i plays the role of the dealer D). Once a player p_i receives enough shares from different players, it moves on to the next stage in which it starts reconstructing the values and chooses its output accordingly.

AVSS from private-coins. The construction of AVSS is quite involved and outside of the scope of this work. But we emphasize that the construction employs private-coins and uses them as the only source of randomness. Therefore, we conclude that private-coins can be used to implement a $\frac{1}{4}$-correct common coin and thus we can achieve efficient Byzantine agreement in the private-coin model.

3.3 Public-Coin

We now move on to agreement in the most challenging setting: the public-coin model. Indeed, Lewko [Lew11] proved that any algorithm for reaching agreement in the public-coin model that *does not* consider the identity of the senders nor the message history *must* have *exponential* expected latency.

For that reason, King and Saia [KS16, KS18] adopted a "forensic accounting" mindset in order to overcome the lower bound of Lewko: their algorithm statistically analyzes the message history in order to uncover coalitions of bad players that try to tilt the outcome of the common coin to a certain value and *blacklists* players in this coalition gradually until the good players can start tossing fair-coins. The resulting algorithm does indeed have a polynomial expected latency, however the resilience is far from optimal[4] and stands at $f < (1.14 \times 10^{-9})n$. The main reason behind the low resilience of [KS16, KS18] is the fact that good players may disagree on which players to blacklist due to the scheduling power of the adversary.

Huang et al [HPZ22] improved King and Saia's method and achieved polynomial expected latency with resilience $f < n/4$. To the best of our knowledge, this is the state of the art in public-coin Byzantine agreement. The innovation of [HPZ22] consists of two improvements that reduce the amount of disagreement between good players: first of all, they use "fractional blacklisting" and second of all they keep on updating the history of messages which drastically reduces the amount of disagreement among good players.

3.3.1 Blacklisting Bad Coalitions in a Synchronous Model.

As a warm-up, we discuss a coin-flipping game in a synchronous setting: in each iteration t, each process p_i flips a local coin[5] $c_i \in \{-1, 1\}$ and the output of the common-coin is determined as $sgn\big(\sum_{i\in[n]} c_i\big)$. The coalition of bad players chooses a value $\sigma(t)$ before iteration t starts and their goal is to make sure that the common-coin outputs $\sigma(t)$, in which case the game goes on for another

[4] the optimal resilience for randomized Byzantine agreement algorithms is also $f < n/3$.

[5] For simplicity, we assume the values are $\{-1, 1\}$ instead of $\{0, 1\}$.

iteration. If the output is $-\sigma(t)$ then the game ends and the bad players lose. Thinking of Bracha's algorithm from Fig. 1: bad players choose $\sigma(t) = -v^*$ and indeed if the common-coin outputs $-\sigma(t)$ then we have agreement in the next iteration of Bracha. The goal of the good players after some T iterations is to identify one bad player and blacklist them, assuming that the game did not end with the bad players losing. Blacklisting is done using statistical tests which we describe next. We denote by $X_i(t)$ the coin-toss of the player i in iteration t and X_i the vector $(X_i(1), \dots, X_i(T))$. We also denote the set of bad players in any given point by B and the good players $G = [n] \setminus B$.

Correlation Test. The correlation between two players p_i, p_j is defined as $\langle X_i, X_j \rangle = \sum_{t\in[T]} X_i X_j$. The first statistical test measures the deviation for every pair of players from the expected correlation of two good players. The *good-good* correlation (correlation between two good players that toss fair coins) is expected to be 0, as the coins are uniform and independent, and by a simple Hoeffding bound (over $t \in [T]$), it holds that w.h.p., the good-good correlation remains within distance $\tilde{O}(\sqrt{T})$ of 0. We also expect that in half of the iterations $t \in [T]$, it holds that $\sigma(t) = -sgn\big(\sum_G X_i(t)\big)$. However, the bad players need to make sure that $\sigma(t) = sgn\big(\sum_{i\in[n]} c_i\big)$, so we expect that in and in these iterations, the bad players need to tilt the sum to the correct value or they lose. From this observation, we derive an equation that allows us to lower bound the sum of the bad-bad correlations and thus lower bound the maximum correlation between two bad players, thus creating a separation between the good-good correlations and bad-bad correlations. This means that if we blacklist the pair p_i, p_j that maximizes $\langle X_i, X_j \rangle$, then we are guaranteed that at least one of them is bad.

Note that we cannot tell anything about good-bad correlations, but that is ok since even if we blacklist one good player and one bad player, the good players will retain their majority.

3.3.2 Blacklisting Bad Coalitions in an Asynchronous Model.

The coin-tossing game in an asynchronous model is much more complicated. We start be explaining the main primitive used to track the coin-tosses of all processes, which King and Saia [KS16, KS18] call a *blackboard*.

Blackboard. A blackboard BB is a matrix with n columns and m rows initially filled with default values ($\perp$s), where each process p_i tries to fill column i with the results of its coin tosses using reliable broadcasts. Since the adversary now has scheduling powers, it can control the rate at which each column is filled and when each value in $BB(i, \cdot)$ reaches a correct process. In addition, since there might be to f uncooperative processes, now correct process can wait for more than $n - f$ columns to be filled. When a correct process sees that BB has $n - f$ columns, it deems it "complete".

For a process p_i, we denote by BB^i the process' of the blackboard BB. Since the adversary controls the scheduling, there might be up to f disagreements between two correct processes. Namely, the last entry of each partial column might be either the broadcast value or $\perp$.

Iterated blackboard. Imagine an algorithm that requires a sequence of blackboards $BB_1, ..., BB_T$. Using the original blackboard of King and Saia, there might be up to f disagreements in *each* blackboard. The innovation of Huang et al [HPZ22] is the use of an *iterated blackboard* to represent a sequence of blackboards, in a way that guarantees a maximum of f disagreements over all of the blackboards. Furthermore, if a correct process p_i receives a valid value in column j in blackboard BB_{t+1}, then by the time it completes BB^i_{t+1}, it will have the same view as p_j for all previous blackboards $BB_1, ..., BB_t$.

Blackboards and weights for coin-tossing. Instead of $X_j(t)$ being the value of the coin tossed by p_j in iteration t, in the asynchronous game we define $X_i(t)$ to be the sum of all entries in $BB(j,\cdot)_t$. Similar to before, we let $X^i_j(t)$ be the view of p_i of the value $X_j(t)$ (based on BB^i_t).

In addition, each process p_i now has a weight w_i which represents how much other processes *trust* p_i. Correct processes maintain complete agreement over the weights for processes that are actively participating in the coin-tossing. This is done using the "furthermore" part in the previous paragraph: when p_j completes BB_{t+1}, let j be some non-empty column in BB_{t+1}, then p_i has an identical view of the previous blackboards to that of p_j and can use that view to calculate the new wight w_j. Since all correct processes will also have that same view, then they will compute the same value of w_j.

In iteration t, the value of the coin flip that p_i outputs is $sgn\big(\sum_{j\in[n] w_j \cdot X_j(t)}\big)$. And similar to the synchronous game, the adversary fist chooses a value $\sigma(t)$ and the game ends naturally (with the bad side losing) if the output of the coin flip is $\sigma(t)$.

Fractional Blacklisting Bad Coalitions. An epoch consists of $T = \tilde{\Theta}(n^2)$ iterations, at the end of which the good players use statistical tests to reduce the weights of (bad) players. Reducing the weights gradually, rather than totally blacklisting "suspects", can be thought of as *fractional blacklisting.* It is proven that after $O(f)$ epochs, w.h.p. all bad players get to weight 0 or the game ends naturally.

There are two statistical tests that determine the fractional blacklisting: the first is a *weighted* correlation test, similar to the simple test we saw before, and the second is deviation from the $l2$ norm for individual players. Namely, processes keep tab of the following quantities:

- $\text{dev}(i) = \sum_{t\in[T]}(w_i X_i(t))^2$
- $\text{corr}(i,j) = \sum_{t\in[T]} w_i w_j X_i(t) X_j(t)$

It is proven using a "Gap Lemma" that if the game does not end naturally, then w.h.p. some bad player p_i will either have a high $\text{dev}(i)$ or will be part of a pair i, j that has a high $\text{corr}(i, j)$.

The exact amount to be reduced from the weights is calculated using a maximal matching algorithm. The graph on which we run the algorithm is a clique (with self loops) s.t. each vertex represents a process and each edge represents the correlation between two processes. After each epoch, each process generates the graph based on its view and sets the capacity of the vertex i to be the weight of p_i, the capacity of a simple edge $\{i, j\}$ depends on the deviation from the expected good-good correlation of p_i, p_j and the capacity of a self-loop (i, i) depends on the deviation of $\text{dev}(i)$ from the expected good $\text{dev}(i)$.

To account for the possible difference of the capacities set by two correct processes (due to different views of BB_t, the maximal matching algorithm used has a Lipschitz property (small difference in the input results in a small difference in the output). This is used to bounded the difference between the weight reductions issued by different correct processes. After the reductions, the weights are rounded down so that it is guaranteed w.h.p. that if some good process thinks that $w_j = 0$ then all good processes think the same thing.

This concludes our discussion of efficient Byzantine agreement in the public-coin model.

4 Connections Between Models

In this section, we demonstrate connections between the different models of randomness. We say that *model A implies model B* if there exists a transformation that takes any A-protocol $\mathcal{A}$ and outputs a B-protocol $\mathcal{A}'$. The latency of $\mathcal{A}'$ should be polynomially related to the latency of $\mathcal{A}$. We require the resilience in both algorithms to be $\Omega(n)$.

Observe that since the transformations operate within the same communication model, we only need to show to handle compute events. To prove the correctness of our transformations, we take inspiration from the *Real/Ideal* paradigm used in *Secure Multi-Party Computation*. Namely, for an A-to-B transformation, we show that if there is an adversary that can make the processes output $y_1, ..., y_n$ in the transformed algorithm in model B, then there exists an adversary that can make the processes output the same $y_1, ..., y_n$ in the original protocol in model A.

We start by discussing the power of the private-coin model, then move on to discussing the relationship between the public-coin model and the common-coin model.

4.1 Private-Coin in relation to other models.

Public-Coin implies Private-Coin. It is trivial that public-coin protocols are at least as strong as private-coin protocols, that is, if we can guarantee safety and liveness against a computationally unbounded adversary from which we cannot keeping secrets, then we can definitely guarantee them against a computationally bounded adversary from which we are allowed to keep secrets. The transformation is, therefore, trivial: simply use the same protocol.

Common-Coin implies Private-Coin The work of Canetti and Rabin [CR93] can be interpreted more broadly than just achieving Byzantine agreement in the private-coin model: thanks to the modular approach in which they presented their construction, we now know a weak common-coin construction in the private-coin model, therefore we conclude that any algorithm that works in the weak common-coin model can be transformed to a private-coin protocol, thus proving that private-coins are at least as strong as a (weak) common-coin.

Is Private-Coin strictly stronger? We have already seen that private-coin is at least as strong as the other models. Intuitively, it seems that private-coin is strictly stronger than public-coin and even arguably common-coin. In order to prove that formally, we would have to present an interface that can be implemented (efficiently) in the private-coin model but not the other models, thus proving a "gap" between the models.

In addition, to ensure the gap is interesting, the interface used for the separation should not require privacy, which trivially cannot be satisfied in the public-coin model. For example, it is easy to show that by definition, the AVSS interface (see Section 3.2) cannot be implemented in the public-coin model, since the interface requires that the adversary cannot learn the secret before at least one correct process invokes the reconstruct procedure Rec. However, if the adversary sees the state of the correct processes (or even just knows the random coins they used to generate the shares of the secret) then it can simulate the correct processes calling Rec and obtain the secret value before the correct processes call Rec.

4.2 Public-Coin implies Strong Common-Coin

The transformation from public-coin to strong common-coin is relatively easy: each process that wants to toss a local random coin, goes through the strong common-coin primitive and thus, effectively, shares the random coin with everyone. The key insight is that since the protocol is public-coin, we don't mind sharing the coin, but the processes must be careful to not request coins before they actually need to use them, because this will give the adversary more power than it does in the public-coin model.

There are subtle points to pay attention to here. For example, if we use a single common-coin primitive then we need to coordinate when each process wants to call it, which would be difficult in an asynchronous system. Instead, we assume we have n such primitives $\{\mathcal{C}_i\}_{i\in[n]}$, where $\mathcal{C}_i$ is responsible for generating the coins for p_i. In addition, the unpredictability of the common-coin is crucial for guaranteeing safety and liveness of the public coin protocol, therefore when p_i wants to generate r random coins, it broadcasts to everyone that it wants to do so, and only then the correct processes make r calls to $\mathcal{C}_i$.

Let $\mathcal{A}$ be the public-coin that we want to transform. The transformation is described in Algorithm 1. Denote the resulting strong common-coin protocol by $\mathcal{A}'$.

Algorithm 1 Public-coin to strong common-coin

```
Code for process $p_i$.
1: procedure OnCompute( )
2:     Move messages from $In_{j\to i}$ to internal buffers $In^*_{j\to i}$.
3:     Let $r$ be an upper bound on the needed random coins for the local computation.
4:     if $r \neq 0$ then Broadcast $(i, r)$ to everyone.
5:     Make $r$ invocations to $\mathcal{C}_i$.
6:     Ignore other compute($i$) events until invocations return.
7:     Once all invocations terminate, process buffers $In^*_{j\to i}$ according to $\mathcal{A}$.
8:     Deposit the new messages to $Out_{i\to j}$ for all $j \in [n]$.
9: procedure OnReceived($j, r$)
10:    Make $r$ calls to $\mathcal{C}_j$.
```

To prove that the transformation in Algorithm 1 preserves safety and liveness, we show that every adversary S' attacking $\mathcal{A}'$ can be simulated by an adversary S attacking $\mathcal{A}$, such that S achieves whatever S' manages to achieve.

Fix an adversary S'. Since we assume deterministic adversaries without loss of generality, it holds that S' schedules an event based on its view which consists of the inputs, all messages (previously delivered or currently in the buffers), the schedule so far and the public-coins it has seen. In the common-coin model, the adversary has the power let a process p_i start its local computation, then make it wait for the random coins that return from $\mathcal{C}_i$. However, since p_i waits until the coins are revealed, the adversary gets the same effect as just delaying the compute(i) event. Since p_i waits for the invocations of $\mathcal{C}_i$, we can assume that S' delivers them all together right before it schedules endCompute(i). We divide the compute events to two types:

- beginCompute(i) entails "preprocessing" messages and invoking $\mathcal{C}_i$.
- endCompute(i) entails receiving the random coins from $\mathcal{C}_i$ and actually processing the messages and performing the local computation accordingly.

We now describe an adversary S that simulates S'. Whenever S wants to schedule an event, it simulates S' using its own view to determine the event to be scheduled:

- If S' schedules a $\mathsf{beginCompute}(i)$, then S just updates the view of S' and schedules $\mathsf{deliver}(j,i)$ for all relevant processes p_j.
- If S' schedules an $\mathsf{endCompute}(i)$, then S schedules $\mathsf{compute}(i)$ and looks at all random coins returned by $\mathcal{C}_i$.
- If S' schedules a $\mathsf{deliver}(i,j)$ between two "regular" processes, then S just updates the view of S' but does not schedule a deliver.

An execution $\mathcal{E}'$ of the algorithm $\mathcal{A}'$ is equivalent to an execution $\mathcal{E}$ of the algorithm $\mathcal{A}$, if the processes have the same inputs $x_1, ..., x_n$ and return the same output $y_1, \ldots, y_n$. For any fixed inputs $x = x_1, ..., x_n$ and coins (infinite tapes) $\rho = \rho_1, \ldots, \rho_n$, we define $\mathcal{E}(x,\rho)$ (resp.$\mathcal{E}(x,\rho)'$) as the execution of $\mathcal{A}$ (resp. $\mathcal{A}'$) on inputs x and ρ_i as the source of randomness for p_i (resp. the values returned by $\mathcal{C}_i$).

Proposition 1. *Fix inputs $x = x_1, ..., x_n$ and infinite tapes $\rho = \rho_1, \ldots, \rho_n$. The, $\mathcal{E}(x,\rho)$ and $\mathcal{E}'(x,\rho)$ are equivalent.*

Proof. Denote by $\mathcal{E}$ (resp. $\mathcal{E}'$) the execution of $\mathcal{A}$ (resp. $\mathcal{A}'$). Let $A_1, A_2, ...$ and $A'_1, A'_2, ...$ denote the scheduling of the events in $\mathcal{E}$ and $\mathcal{E}'$ respectively. For the execution $\mathcal{E}'$, we can imagine that each process p_i simulates a node p'_i running the original protocol $\mathcal{A}$: its incoming messages are stored in $In'_{j\to i}$ and its randomness is supplied by $\mathcal{C}_i$. We say that events A_ℓ and A'_k are equivalent if the state (including the buffers) of each process p_i in A_ℓ is identical to the state of the internal process p'_i in A'_k and the view of S' in A'_k is the same as the view S simulates for S' in A_ℓ. The takeaway is that if A_ℓ and A'_k are equivalent, then simulating S' on the view in either event yields the same result, i.e., scheduling the same next event. We show a non-descending mapping $\phi : \mathbb{N} \to \mathbb{N}$ s.t. $A_{\phi(k)}$ and A'_k are equivalent by building it inductively. The base is pretty simple: initially, all processes have the same inputs (by assumption), therefore the states are identical. Now for the step, assume we have successfully mapped k to $\ell := \phi(k)$. Consider the event A'_{k+1}. It follows that S' scheduled A'_{k+1} based on the view it had after scheduling A'_k. By the construction of S, it simulates S' on the view of A_ℓ which, by the induction hypotheses, is the same as A'_k. The result of the simulation is scheduling the same event A'_{k+1}, according to which S may or may not schedule the next event.

- If A'_{k+1} is a $\mathsf{deliver}(i,j)$ event then S does not actually schedule an event, but only updates the view of S' to see that $\mathsf{deliver}(i,j)$. The buffer $In_{j\to i}$ does not change in $\mathcal{E}$, but neither does the internal buffer $In'_{j\to i}$ in $\mathcal{E}'$ so the states do not really change, and we set $\phi(k+1) = \ell$.
- If A'_{k+1} is a $\mathsf{beginCompute}(i)$ event then S schedules the relevant $\mathsf{deliver}(j,i)$, in order to emulate the process p_i moving the messages to the internal buffer $In'_{j\to i}$. Since the states were identical in A_ℓ and A'_k, we get that the content of $In_{j\to i}$ in $\mathcal{E}$ is the same as the content of $In'_{j\to i}$ in $\mathcal{E}'$. We assume that all $\mathsf{deliver}(j,i)$ events are batched into one and set $\phi(k+1) = \ell+1$.
- If A'_{k+1} is a $\mathsf{endCompute}(i)$ event then $A_{\ell+1}$ is a $\mathsf{compute}(i)$ event, and we set $\phi(k+1) = \ell+1$. Note that the state of the process p_i in A_ℓ is identical to the state of p'_i in A'_k and furthermore,

the random coins returned by $\mathcal{C}_i$ in $\mathcal{E}'$ match the local (public) coin tosses in $\mathcal{E}$ (which are determined by ρ_i), so we get that p_i performs the same local computation (reaches the same state and deposits the same messages).

By applying this induction to the each event in which some process p_i terminates and returns a value, we get that processes return the same value in both executions, and this concludes the proof. ■

We move on to proving the main result of this section, summarized in the following lemma:

Lemma 2. *All safety and liveness properties of $\mathcal{A}$ are inherited by $\mathcal{A}'$.*

Proof. For any fixed inputs $x = x_1, ..., x_n$, we define the random variable $\mathcal{E}(x)$ as $\mathcal{E}(x, \rho)$ where ρ is uniformly distributed over $\{0, 1\}^*$. We define $\mathcal{E}(x)$ similarly.

Finally, we utilize the fact that $\mathcal{C}_i$ is a *strong* common-coin, so the values it returns are independently and uniformly distributed over $\{0, 1\}$, just like the local coins of a good process. By Proposition 1, we know that $\mathcal{E}(x, \rho)$ and $\mathcal{E}'(x, \rho)$ are equivalent and by the identical distribution of ρ for both algorithms, we get that the outputs of the processes in $\mathcal{E}(x)$ and in $\mathcal{E}'(x)$ are identically distributed, thus proving that the adversary S achieves whatever S' manages to achieve and therefore the safety and liveness properties of $\mathcal{A}'$ follows from the safety and liveness properties of $\mathcal{A}$. ■

4.3 A Common-Coin without Secrets?

It is tempting to interpret the work on public-coin Byzantine agreement [KS16, KS18, HPZ22] as achieving common-coin in the public-coin model. However, we observe that the algorithm we of [HPZ22], which we discussed in Section 3.3, does not quite construct a common-coin in the standard sense. However, we stress that this is not the case; namely, the correctness property, as defined in Section 2.2.1, does not hold in for their coin-tossing protocol using blacklisting. The only guarantee is that if the adversary tries to influence too many iterations, then the corrupted players will be blacklisted (little by little).

It is unclear how one can extend this approach to construct a common-coin in the public-coin model, because the adversary can get away with cheating in a small number of iterations. The naive approach of conducting a large number of iterations then somehow "extracting" a good iteration out of them seems to fail since it is enough for the adversary to influence a small number of iteration to affect the final result. For example, if we want the output to be the XOR of all previous iterations, then it's enough for the adversary to cheat in the last iteration in order to influence the output. Similarly, if we choose the "effective" iteration randomly at the end, then the adversary only needs to cheat in the last iteration to sway the good processes to his desired iteration.

Even a leader election approach would not work in this model due to the adaptive nature of the adversary: it can simply wait for the leader to be elected and then corrupt only it.

Amplifying Common-Coins. Another interesting question in its own right is whether we can transform a weak common-coin to a strong common-coin, even with the help of local (public) randomness. We can even relax the definition of "strong" and require that the agreement parameter is $\varepsilon = \frac{1}{2} - \text{negl}(n)$ where $\text{negl}(n) = o(n^{-c})$ for every constant c.

References

[ADS20] Ittai Abraham, Danny Dolev, and Gilad Stern. Revisiting asynchronous fault tolerant computation with optimal resilience. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 139–148. ACM, 2020.

[BKLL20] Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part I*, volume 12550 of *Lecture Notes in Computer Science*, pages 353–380. Springer, 2020.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[CR93] 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.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[HPZ22] 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.

[KKK$^+$10] Bruce M. Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. *ACM Trans. Algorithms*, 6(4):68:1–68:28, 2010.

[KS16] Valerie King and Jared Saia. Byzantine agreement in expected polynomial time. *J. ACM*, 63(2):13:1–13:21, 2016.

[KS18] Valerie King and Jared Saia. Correction to byzantine agreement in expected polynomial time, JACM 2016. *CoRR*, abs/1812.10169, 2018.

[Lew11] Allison B. Lewko. The contest between simplicity and efficiency in asynchronous byzantine agreement. In David Peleg, editor, *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2011.

[MMR15] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with t < n/3, o(n2) messages, and O(1) expected time. *J. ACM*, 62(4):31:1–31:21, 2015.

[Rab83] Michael O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 403–409. IEEE Computer Society, 1983.