SureDistrib: Verifying Almost-Sure Termination of Composite Asynchronous Byzantine Protocols

LONGFEI QIU, Yale University, USA
JINGQI XIAO, University of Hong Kong, China
JI-YONG SHIN, Northeastern University, USA
ZHONG SHAO, Yale University, USA

Consensus algorithms play a central role in many distributed systems, including blockchains. The most practical consensus algorithms are based on the partial synchrony model. While partially synchronous protocols are relatively simple, they cannot maintain liveness when the message delivery latency is uncertain. Asynchronous protocols do not rely on bounded latency to maintain liveness, but they are much more difficult to understand and implement, being usually described as a composition of several layers of algorithms. Moreover, due to the FLP impossibility theorem, they only provide probabilistic liveness guarantees. These factors make the correctness of asynchronous protocols challenging to verify, and there have been liveness bugs in these protocols that remain unnoticed for years.

We introduce SureDistrib, a formal framework for specifying and verifying probabilistic safety and liveness properties of asynchronous distributed protocols. Our framework supports specifying probabilistic algorithms that depend on other probabilistic functionalities, such as binary agreement algorithms depending on common coins. We define refinement relations for such systems, and prove composition lemmas that replace the underlay functionality with an implementation, so that the composed system refines the original system with an abstract underlay. Based on our framework, we give the first mechanized proof that an asynchronous byzantine fault-tolerant binary agreement algorithm terminates with probability 1 ("almost-sure termination").

CCS Concepts: • **Networks** → **Protocol testing and verification**; • **Theory of computation** → **Distributed algorithms**; • **Mathematics of computing** → **Probabilistic algorithms**.

Additional Key Words and Phrases: byzantine fault-tolerance, asynchronous consensus, formal verification, almost-sure termination, Rocq proof assistant

ACM Reference Format:
Longfei Qiu, Jingqi Xiao, Ji-Yong Shin, and Zhong Shao. 2026. SureDistrib: Verifying Almost-Sure Termination of Composite Asynchronous Byzantine Protocols. *Proc. ACM Program. Lang.* 10, PLDI, Article 215 (June 2026), 31 pages. https://doi.org/10.1145/3808293

1 Introduction

Blockchains are large-scale distributed systems that run upon thousands of server nodes communicating over the open Internet. They must maintain robustness in the face of a diverse range of adverse network conditions, such as byzantine attacks and uncertain network delays. While early blockchains like Bitcoin [51] used Proof-of-Work (PoW) to determine the transaction log, most modern blockchains use Proof-of-Stake (PoS), where a dynamic set of validator nodes participate in a byzantine fault-tolerant (BFT) consensus protocol to build the transaction log. The security of the blockchain thus depends on the safety and liveness of the BFT protocols.

Authors' Contact Information: Longfei Qiu, Yale University, New Haven, USA, longfei.qiu@yale.edu; Jingqi Xiao, University of Hong Kong, Hong Kong, China, u3013038@connect.hku.hk; Ji-Yong Shin, Northeastern University, Boston, USA, j.shin@northeastern.edu; Zhong Shao, Yale University, New Haven, USA, zhong.shao@yale.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART215
https://doi.org/10.1145/3808293

Every message-passing distributed system depends on getting messages delivered to make progress. A key factor in the design of distributed protocols is the assumption one makes on the message delivery latency. The most practical BFT protocols [15, 32, 73] are all based on the so-called *partial synchrony* model [23]. Under this model, the network system may alternate between periods of synchrony, where there is a known bound Δ on the message delivery latency, and periods of asynchrony, where messages may arrive arbitrarily late, hence the name "partial synchrony." The goal of the protocol is to maintain safety regardless of network latency, but provide liveness guarantees only during periods of synchrony.

While partially synchronous BFT protocols have worked remarkably well in practice, their inability to guarantee liveness during periods of asynchrony has often been viewed as a sore point. This has led to a long line of works [3, 8, 12, 21, 29, 34, 40, 49] proposing BFT protocols that guarantee liveness under asynchrony. Despite that significant progress made in the theoretical design of asynchronous protocols, these protocols have remained little adopted in the industry. We believe this situation is due to a combination of two factors.

First, there is a theoretical limitation on what asynchronous protocols can provide. By the well-known FLP impossibility theorem [28], no deterministic consensus protocol can maintain both safety and liveness under asynchrony even with only one faulty process. Therefore, all asynchronous protocols must exploit randomness to provide probabilistic liveness guarantees.

Second, compared to the partially synchronous protocols, asynchronous protocols are notably more complex to understand and implement. Whereas partially synchronous protocols usually have simple structures and are described monolithically, asynchronous protocols are often described as a composition of several layers of algorithms. For example, the HoneyBadger protocol [49] is defined in three layers: a binary agreement (BA) layer, an asynchronous common subset (ACS) layer, and an encryption layer for avoiding censorship. This increased degree of complexity, combined with the probabilistic nature of these algorithms, makes it very easy to introduce bugs that are difficult to discover through manual review or conventional testing techniques.

In fact, as we will see later, the HoneyBadger protocol [49] contains a very subtle liveness bug [48]. The bug was not noticed until a year after HoneyBadger was published, and years after its discovery, there are still controversies regarding the fix [72]. These controversies highlight the limitations of on-paper proofs for safeguarding the correctness of asynchronous protocols.

The probabilistic and compositional nature of asynchronous consensus protocols also poses serious challenges to formally verifying their correctness. In the realm of partial synchrony, it has been observed that almost every consensus protocol can be interpreted as a periodically rotated leader driving the growth of a consensus tree, leading to formal models such as QTree [17], LiDO [54], and Trees & Turtles [52]. Even very complex protocols can be formally verified by showing a contextual refinement to these abstract models [55]. By contrast, formally verifying or even just specifying the liveness properties of asynchronous protocols, such as "the processes eventually reach agreement on a single value with probability 1," remains challenging. These difficulties hinder the development of mechanized correctness proofs for asynchronous protocols.

In this work, we study the problem of formally verifying the probabilistic safety and liveness properties of asynchronous consensus protocols. We introduce SureDistrib, a compositional framework for specifying and proving probabilistic safety and liveness properties of distributed systems. We focus on systems with *static corruption* and where the network adversary $\mathcal{A}$ always has *full knowledge* of the system state, but each process in the system including the adversary can make probabilistic moves. We use the "adversary" as an abstraction for all non-deterministic events in the system that are beyond the control of honest processes, including message delivery and byzantine process actions. By "full knowledge" we mean the adversary can inspect the content of every network packet and the internal state of every process, and can behave adaptively according

to this knowledge. While this model excludes certain protocols that assume network messages are opaque to the adversary (e.g. [29]), we will see that a large class of asynchronous protocols (e.g. [21, 34, 40, 49]) that use common coins [58] as the sole source of randomness can be analyzed with our framework, provided the common coin is modeled as a black box primitive. Thus the SureDistrib framework can be interpreted as a small fragment of the I/O automata theory [46, 71]: although it lacks many advanced features of I/O automata, it is sufficient for the most practical kinds of probabilistic distributed algorithms. On the flip side, the simplicity also makes the theory easy to mechanize in a proof checker like Rocq [68].

Based on our framework, we formally specify in Rocq [68] abstract models of common coins, graded crusader agreement (GCA), and binary agreement (BA). We adapt the *segmented trace* technique of Qiu et al. [54] to the asynchronous and probabilistic setting to specify the liveness properties of these functionalities. We provide formally verified implementations of binary GCA and BA (Fig. 1). Our implementations of binary GCA and BA are loosely modeled upon the BA algorithm from Mostefaoui et al. [50] used in HoneyBadger [49], including the fix for its liveness bug. We prove almost-sure termination of BA by analyzing the expectation of a state variable of the transition system, and proving the variable is a submartingale (i.e. its expectation never decreases). To our knowledge, this is the first foundational proof that an asynchronous agreement algorithm terminates with probability 1.

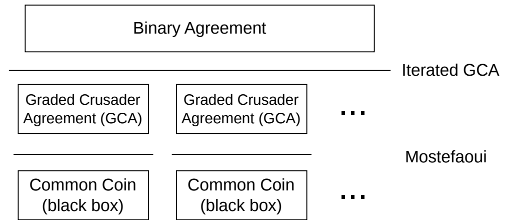

Fig. 1. Summary of our verification results. We implemented graded crusader agreement (GCA) from common coins following an algorithm from Mostefaoui et al. [50]. We also implemented binary agreement (BA) using an infinite sequence of GCA instances.

To summarize, our contributions are:

- **SureDistrib**, a compositional framework for specifying and implementing probabilistic distributed systems;
- **Formal models** of three asynchronous functionalities under the SureDistrib framework: common coins, graded crusader agreement (GCA), and binary agreement (BA);
- **Implementations** of binary GCA and BA with formally verified safety and liveness proofs.
- **Almost-sure termination proof** of BA by analyzing the expectation of a state variable.

Our mechanized proofs are available as an artifact [56]. We provide some more technical details in our extended technical report [57].

2 Overview

2.1 Motivation: The Liveness Bug of HoneyBadger

Our analysis begins with an asynchronous consensus algorithm called HoneyBadger [49]. The HoneyBadger protocol implements a functionality known as *atomic broadcast* [19]. Informally, an atomic broadcast protocol provides each process with an API called `Input(tx)`. Each process may call `Input(tx)` multiple times, each time submitting a transaction `tx`. Each process outputs a sequence of transactions, and all honest processes must output the same sequence. Also, transactions submitted by honest processes must eventually appear in the output. Thus atomic broadcast is the asynchronous counterpart of what partially synchronous algorithms like HotStuff [73] provides.

Throughout this work we assume a standard $3f + 1$ network configuration: there are a total of $3f + 1$ processes and the adversary may statically corrupt f processes at the beginning of execution. The corrupted processes are called *byzantine* and the remaining processes are called *honest*. A

quorum is any set of at least $2f + 1$ processes. We also call a set of at least $f + 1$ processes a *blocking set*. Every quorum has a blocking subset with only honest processes. A quorum and a blocking set always intersects on at least one member.

The HoneyBadger protocol has a three-layered structure (Fig. 2). The lowest layer is a BA protocol. The BA functionality allows each process to input a single bit $b \in \{0, 1\}$. All honest processes eventually agree on a single output bit b', which must be the input of at least one honest process. More precisely, the BA algorithm needs to satisfy:

- **Agreement**: If one honest process outputs b, then the output of every honest process is b.
- **Validity**: If one honest process outputs b, then b is the input of an honest process;
- **Probabilistic Termination**: After all honest processes input a bit, with a probability that converges to 1 over time, all honest processes eventually output a bit.

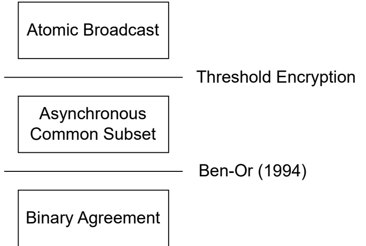

Fig. 2. The three-layered structure of HoneyBadger. The lowest layer is a BA protocol from Mostefaoui et al. [50]. It implements ACS upon BA using an algorithm from Ben-Or et al. [9]. Finally, it implements atomic broadcast upon ACS using threshold encryption. The purpose of encryption is to avoid censorship from the adversary.

The HoneyBadger protocol then implements asynchronous common subset (ACS) upon $3f + 1$ instances of BA. The ACS protocol allows each process to submit an input from some domain of bit-strings. Each process eventually receives the inputs from at least $2f + 1$ processes, and every honest process receives the same set of inputs. Thus each process can take the received set of inputs, ordered by process ID, as the sequence of committed transactions to output. The idea of ACS is to let each process reliably broadcast its input, which prevents equivocation of byzantine processes. Upon receiving the input of process p_i via reliable broadcast, each process inputs 1 to the i-th instance of BA. After receiving an output of 1 from at least $2f + 1$ instances of BA, each process inputs 0 to the BA instances that it has not input a value, to ensure termination of all instances. HoneyBadger also employs encryption to avoid censorship from the adversary.

From the above description it is immediately clear that liveness of HoneyBadger depends on liveness of ACS, which in turn depends on the conjunction of $3f + 1$ instances of BA. If any single instance of BA gets stuck, the protocol cannot make progress. Unfortunately, the BA algorithm used by HoneyBadger has a liveness bug.

The BA Algorithm. The BA algorithm used by HoneyBadger is adapted from Mostefaoui et al. [50] and we summarize it in Alg. 1. Like partially synchronous algorithms, the BA protocol proceeds in *rounds*, and every step is associated with a round number r, starting from 0. The algorithm depends on an infinite sequence of common coins, also indexed by round numbers. Each instance of the common coin provides a single API called `OpenCoin()`, which is a request to reveal the value of the coin. The coin value is a single uniformly random bit, and it is not revealed until $f + 1$ processes have called `OpenCoin()`. After the coin value is revealed, each honest process eventually receives the same bit, provided the process has called `OpenCoin()` previously.

The protocol uses two kinds of messages called `BVAL` and `AUX`. Each message has an *id* field for the sender ID, an r field for the round number, and a b field that carries a single bit. The underscore is a placeholder for fields whose value we do not care. On line 7, id' denotes the sender ID. The prime distinguishes it from *id* which is ID of the current process.

An intuitive explanation of the algorithm is as follows. At the beginning of each round, the processes need to find values which are valid (submitted by at least one honest process). They do so by broadcasting `BVAL` messages for values they believe are valid. If a blocking set of processes have sent `BVAL` for a particular bit b (line 3), then at least one honest process have sent `BVAL` for b,

so b is certainly valid. Furthermore, if a quorum of processes have sent BVAL for b (line 5), then at least $f + 1$ honest processes have sent BVAL for b, and everyone will eventually learn that b is valid. At this point, the process sends out an AUX message for b, which can be interpreted as a request to commit b in this round. Notice that within each round, an honest process may send out multiple BVAL messages, but will only send out one AUX message (line 5). Hence two sets of AUX messages, each from $2f + 1$ processes, must intersect on at least one member.

Algorithm 1 BA Protocol from Mostefaoui et al. [50]

1: **upon** receiving input bit b:
2: Broadcast $\langle \mathsf{BVAL}, id, 0, b \rangle$. ($id$ is process ID)
3: **upon** receiving $\langle \mathsf{BVAL}, _, r, b \rangle$ for a particular (r, b) from $f + 1$ processes, and this process did not broadcast $\langle \mathsf{BVAL}, id, r, b \rangle$ previously:
4: Broadcast $\langle \mathsf{BVAL}, id, r, b \rangle$.
5: **upon** receiving $\langle \mathsf{BVAL}, _, r, b \rangle$ for a particular (r, b) from $2f + 1$ processes, and this process did not broadcast $\langle \mathsf{AUX}, id, r, b' \rangle$ for any b' previously:
6: Broadcast $\langle \mathsf{AUX}, id, r, b \rangle$.
7: **upon** receiving $\langle \mathsf{AUX}, id', r, b \rangle$ for a particular (id', r, b), and the process has previously received $\langle \mathsf{BVAL}, _, r, b \rangle$ from $2f + 1$ processes:
8: Mark $\langle \mathsf{AUX}, id', r, b \rangle$ as valid.
9: **upon** observing valid $\langle \mathsf{AUX}, _, r, _ \rangle$ for a particular r from $2f + 1$ processes:
10: Call OpenCoin() on the r-th common coin.
11: **upon** receiving the value of the r-th common coin:
12: $B \leftarrow$ value of the r-th common coin.
13: $vals \leftarrow \{b \mid \text{valid } \langle \mathsf{AUX}, _, r, b \rangle \text{ received}\}$.
14: **if** $vals = \{B\}$ **then**
15: Output B.
16: Broadcast $\langle \mathsf{BVAL}, id, r + 1, B \rangle$.
17: **else if** $vals = \{0, 1\}$ **then**
18: Broadcast $\langle \mathsf{BVAL}, id, r + 1, B \rangle$.
19: **else** ▷ $vals$ is non-empty, so $vals = \{1 - B\}$
20: Broadcast $\langle \mathsf{BVAL}, id, r + 1, 1 - B \rangle$.

To check whether b can be actually committed in this round, a process needs to wait for a quorum of AUX messages (line 9). If it receives $2f + 1$ AUX messages for the same bit b, then every honest process will receive at least one AUX message for b, due to the intersection property above, so b can be potentially committed, but it cannot be sure whether AUX messages for the other bit exist. In the more general case, the process receives AUX messages for both values $b \in \{0, 1\}$, and a decision cannot be made. Here the common coin kicks in: it supplies a bit B such that if any honest process commits a bit b in this round (line 15), then $b = B$. With this promise, we proceed as follows. If we observe AUX for B from $2f + 1$ processes (line 14), then we say B is "committed" in this round, and output B to the caller (line 15). If we observe AUX for both possible values (line 17), then we cannot be certain whether some other process will commit B in this round. In this case we broadcast BVAL for B in the next round. We say B is "maybe committed" in this case. Finally, if we don't observe any AUX for B (line 19), then we are certain no process will commit B in this round. We say B is "not committed" in this case.

Agreement of BA relies on the following observation. If some honest process commits B in round r, then in round $r + 1$ all honest processes will broadcast BVAL only for B. By induction, we see that only B can be committed by other processes in any round $r' > r$. Liveness of BA requires that some honest process will eventually commit a bit B.

The Liveness Bug. Recall (see Sec. 1) that in our system model the network adversary $\mathcal{A}$ always has full knowledge of the system. It turns out that under this model, there is a way for the adversary to prevent Alg. 1 from reaching convergence [48]. We assume that at least one honest process has broadcast $\langle \mathsf{BVAL}, _, 0, 0 \rangle$, and another honest process has broadcast $\langle \mathsf{BVAL}, _, 0, 1 \rangle$. If each byzantine process broadcasts both $\langle \mathsf{BVAL}, _, 0, 0 \rangle$ and $\langle \mathsf{BVAL}, _, 0, 1 \rangle$, then each honest process will execute line 4 of Alg. 1, and we get both $\langle \mathsf{BVAL}, _, 0, 0 \rangle$ and $\langle \mathsf{BVAL}, _, 0, 1 \rangle$ from $2f + 1$ processes.

Now we segregate the honest processes into two sets X and Y with $|X| = f + 1, |Y| = f$. We deliver the BVAL messages to processes in X, so they will broadcast AUX messages. By controlling the delivery order we can get both at least one $\langle \mathsf{AUX}, _, 0, 0 \rangle$ and at least one $\langle \mathsf{AUX}, _, 0, 1 \rangle$. We also let each byzantine process broadcast an AUX message, so we have a total of $2f + 1$ AUX messages. We deliver these AUX messages to processes in X, so they will call OpenCoin() and we can reveal the coin value. (Technically, we only need one honest process to call OpenCoin() to get over the $f + 1$

threshold, but the point here is Alg. 1 fails even if we increase the threshold to $2f + 1$.) Regardless of the coin value B, each process in X will believe B is "maybe committed," because they observe AUX messages for both bits. These processes will broadcast BVAL for B in the next round (line 18).

Now the liveness attack is to let the remaining honest processes, i.e. the processes in Y believe that B is "not committed." They will broadcast BVAL for $1-B$ in the next round (line 20). By repeating this process for each round, the algorithm will never converge. Recall that regardless the value of B, there is at least one AUX for $1 - B$ from the processes in X. We deliver the BVAL messages for $1 - B$ to the processes in Y, so that all of them will broadcast AUX for $1 - B$. Also, we let each byzantine process broadcast an AUX message for $1 - B$. Together, this gives us $2f + 1$ AUX messages for $1 - B$. If we deliver these AUX messages to processes in Y, they will consider B as "not committed."

Since its discovery, the liveness bug of Alg. 1 has been analyzed by several works [2, 30]. Abraham et al. [2] summarized the issue as the failure of a *binding* property of each round: by the time the common coin is revealed, there should be a bit b such that every process will receive at least one AUX message for b. However, Alg. 1 allows the adversary to choose which AUX messages each honest party receives based on the coin value.

The Takeaway. Without following all the details of HoneyBadger in the previous pages, we can make some general observations about asynchronous consensus protocols. First, these systems are both probabilistic and nondeterministic. The nondeterminism comes from the large number of events occurring concurrently in the system: message delivery, local processing at each process, and byzantine actions. "Liveness" of such algorithms is thus a game between the system and the adversary, and the liveness bug is a strategy for the adversary to prevent the system from reaching an expected state (some process commits a bit) indefinitely. Therefore, to prove liveness of asynchronous consensus we have to quantify over all possible strategies of the adversary, and place a lower bound on the probability of good events occurring in the resulting distribution of states.

Second, these protocols have an inherently compositional structure, as shown by the layered structure of HoneyBadger (Fig. 2). Even the BA algorithm (Alg. 1) should not be considered monolithically: what happens within each round is sufficiently complex that merits studying independently. As we will see, in the distributed system literature, what happens within each round of Alg. 1 is often called Graded Crusader Agreement (GCA) [2], and Alg. 1 can be interpreted as a loop running iterated instances of GCA. Therefore any liveness guarantee we prove about these systems must be presented in a way that can be easily reused by overlay applications. These factors make asynchronous protocols highly challenging to formally model and verify. We now look at how the SureDistrib framework handles these challenges.

2.2 Formal Modeling of Agreement Functionalities under SureDistrib

Probabilistic Transition Systems. Under SureDistrib we model the asynchronous functionalities and their implementations as probabilistic transition systems (PTS) with additional structures. Here we introduce the basic notions of PTS. We characterize a PTS by a tuple $(S, s_0, \delta_d, \delta_p)$, where:

- S is a set of possible states of the system. We represent state distributions of the system by values of type list $(S \times \mathbb{Q})$. Within this work, we shall assume all probabilities are rational. If e is an element in a distribution D, we shall write $e.\mathrm{s}$ for its state part, and $e.\mathrm{p}$ for its probability part. We say a distribution D is *valid*, if $\forall e \in D, e.\mathrm{p} \geq 0$. We say D is *normalized*, if $\sum_{e \in D} e.\mathrm{p} = 1$. When analyzing liveness of protocols, we often consider the system evolution along different branches of a distribution, which are valid but non-normalized distributions. We write $\mathtt{dist}_S$ for the set of valid distributions over S.
- $s_0 \in S$ is the initial state of the system.

- $\delta_d \in \mathcal{P}(S \times S)$ is a non-probabilistic transition relation. Informally, δ_d represents the possible steps of the system, with any randomness provided as an explicit paramter. Even when analyzing probabilistic systems, most invariants are still non-probabilistic. Therefore, it is easier to prove them by approximating the system with δ_d. We write $s \rightarrow_d s'$ if $(s, s') \in \delta_d$. We write $s \rightarrow_d^* s'$ if s' is reachable from s through zero or more steps in δ_d.
- $\delta_p \in \mathcal{P}(S \times \mathtt{dist}_S)$ is a probabilistic transition relation. We require that if $(s, D) \in \delta_p$, then D is normalized, and for every $e \in D$ we have $s \rightarrow_d^* e.\mathrm{s}$. Thus when defining transition systems on paper, we can just define δ_p, and δ_d is implicitly defined by taking all possible transitions in δ_p. In the artifact it is the other way round: we first define δ_d and then δ_p.

The semantics of a PTS is a non-probabilistic transition system on $\mathtt{dist}_S$. The initial state is $[(s_0, 1)]$. There are two kinds of steps, which we call "split" and "action" steps (Fig. 3). In a split step an element e of the distribution is split into multiple elements, all having the same state as e, and the probabilities of these elements sum up to $e.\mathrm{p}$. In an action step a transition in δ_p is executed upon an element e, and e is replaced by the resulting distribution. If two PTS denoted by A, B execute in parallel, then a probabilistic action in A that does not affect B would be a split step from the perspective of B. Also, if the adversary interacting with the PTS makes an internal probabilistic choice, it would be a split step from the perspective of the PTS. We write $D \rightarrow_p D'$ if D' is reachable from D with one split or action step. We write $D \rightarrow_p^* D'$ if D' is reachable from D with zero or more split or action steps.

$$\frac{D = d_1 +\!\!+ [e] +\!\!+ d_2, \quad 0 \le p_1, \cdots, p_n \le 1, \quad \sum_i p_i = 1}{D \rightarrow_p d_1 +\!\!+ [(e.\mathrm{s}, e.\mathrm{p} \cdot p_1); \cdots; (e.\mathrm{s}, e.\mathrm{p} \cdot p_n)] +\!\!+ d_2}\ \text{Split}$$

$$\frac{D = d_1 +\!\!+ [e] +\!\!+ d_2, \quad (e, D') \in \delta_p, \quad D' = [(s_1, p_1); \cdots; (s_n, p_n)]}{D \rightarrow_p d_1 +\!\!+ [(s_1, e.\mathrm{p} \cdot p_1); \cdots; (s_n, e.\mathrm{p} \cdot p_n)] +\!\!+ d_2}\ \text{Action}$$

Fig. 3. Split and action steps.

It is easy to see if D is valid and $D \rightarrow_p^* D'$ then D' is also valid; if D is normalized then D' is also normalized. By induction on probabilistic steps, we can get the following lemma:

Lemma 1. *If $D \rightarrow_p^* D'$ and $D = [e_1; \cdots; e_n]$, then there exist $D'_1, \cdots, D'_n$ such that $D' = D'_1 +\!\!+ \cdots +\!\!+ D'_n$ and $[e_i] \rightarrow_p^* D'_i$ for each $1 \le i \le n$.*

In other words, each element of D is a probabilistic system that evolves on its own right. This lemma will be important when we formalize the liveness assumptions later.

We will work with expectations of state functions when proving liveness. If f is a function $S \mapsto \mathbb{Q}$ and $D \in \mathtt{dist}_S$ then we define $\mathbb{E}_D[f] = \sum_{e \in D}(f(e.\mathrm{s}) \cdot e.\mathrm{p})$.

Interpreting Alg. 1: The Graded Crusader Agreement Functionality. As shown in Sec. 2.1, although both asynchronous and partially synchronous consensus algorithms have a round-based structure, what happens within each round of asynchronous agreement is notably more complex than partially synchronous algorithms. The goal of this work is to verify liveness of a modified version of Alg. 1. To this end, we will not specify Alg. 1 as a monolithic transition system, as was done in, e.g., IronFleet [36] for Multi-Paxos and LiDO [54] for Jolteon.

The general structure of our interpretation of Alg. 1 is shown in Fig. 1. We capture each round of Alg. 1 with an independent functionality called Graded Crusader Agreement (GCA). The BA algorithm is then a loop that executes iterated instances of GCA. As such GCA has a central position in this work. Here we first introduce GCA semi-formally, and then explain how it is encoded in our SureDistrib framework.

In the distributed computing literature, GCA is a concept with many names (e.g. Graded Broadcast [25], Gradecast [22], Adopt-Commit [6]) and characterizations. The general theme is that each process inputs a value from some domain V, and receives a *graded value* as outcome. The outcomes

received by honest processes do not need to be the same as each other. Typically there are three kinds of graded values, and we denote them by `Committed`(v), `MaybeCommitted`(v), and `NotCommitted`. In the `Committed` and `MaybeCommitted` case the caller receives a value $v \in V$. In the `NotCommitted` case the caller receives nothing. In Sec. 2.1, we hinted at these outcomes when introducing the intuition of Alg. 1. In other works the outcomes of GCA are denoted by (v, x), where $v \in V \cup \{\bot\}$ and $x \in \{0, 1, 2\}$ is a *grade value*. Grade 2 corresponds to our `Committed`(v); grade 0 corresponds to our `NotCommitted`. Here we adopt the more intuitive names for these grades.

In Alg. 2 we describe our abstract model of GCA. We use G_V to denote the set of graded values with domain V. Our model can be summarized by the following properties.

Validity votes and proposals: Each process may call `AddValidVote`(v) to add a *validity vote* for v. It may call `Propose`(v) to add a *proposal* for v. Honest processes may call `AddValidVote`(v) for multiple values, but may only call `Propose`(v) once (line 13). A proposal is not considered *valid* until there are $2f + 1$ validity votes for it. Otherwise byzantine processes can simply propose and commit anything they want. Notice the analogy between validity votes and `BVAL` messages, and between proposals and `AUX` messages in Alg. 1.

The Global Outcome: After $2f+1$ processes have submitted valid proposals, the adversary may set the global outcome of this instance of GCA, which is a value in G_V. However, it cannot set the global outcome arbitrarily. Instead, it must call `OpenGlobalOutcome`(*choice*) with *choice* $\in \mathcal{I}$, and the global outcome is randomly drawn from a distribution D(*choice*) (line 24; $\mathcal{I}$ and D(*choice*) are implementation-specific). This corresponds to the instant where the value of the common coin is revealed in Alg. 1. The *choice* value captures any influence the adversary may have on the distribution of the final outcome. One of the key assumptions of the liveness proof of BA is that, there is a lower bound p_0 on the probability of the global outcome becoming `Committed(_)` regardless of *choice*, and the GCA implementation needs to satisfy this condition. As we explain below, this corresponds to requiring GCA to be *probabilistically binding*, a relaxed version of the binding property of Abraham et al. [2].

Algorithm 2 An abstract model of GCA

1: Implementation-specific parameters:
2: $\quad V$, the domain of input values.
3: $\quad \mathcal{I}$, a set of *adversary choices* which the adversary provides when deciding the global outcome of GCA.
4: $\quad D(choice)$, a function that maps each choice in $\mathcal{I}$ to a normalized distribution on G_V.
5: State variables:
6: $\quad validity_votes$: list$(ID \times V)$.
7: $\quad proposals$: list$(ID \times V)$.
8: $\quad global_outcome$: option G_V.
9: $\quad local_outcomes$: FinMap$(ID \mapsto G_V)$.
10: **upon** p_{id} calls `AddValidVote`(v):
11: $\quad validity_votes \leftarrow (id, v) :: validity_votes$.
12: **upon** p_{id} calls `Propose`(v):
13: $\quad$ **if** p_{id} is honest and has already called `Propose`(v') for some v' **then**
14: $\quad\quad$ **return** *InvalidOperation*.
15: $\quad proposals \leftarrow (id, v) :: proposals$.
16: **upon** p_{id} calls `Learn`():
17: $\quad$ Wait until $local_outcomes[id]$ = Some(t)
18: $\quad$ **return** t.
19: **upon** $\mathcal{A}$ calls `OpenGlobalOutcome`(*choice*):
20: $\quad$ **if** *proposals* does not contain valid (having $2f + 1$ validity votes) proposals from $2f + 1$ processes **then**
21: $\quad\quad$ **return** *InvalidOperation*.
22: $\quad$ **if** $D(choice)$ is not an allowed global outcome distribution **then**
23: $\quad\quad$ **return** *InvalidOperation*.
24: $\quad t \xleftarrow{\$} D(choice)$. ($\xleftarrow{\$}$ means random sampling)
25: $\quad global_outcome \leftarrow$ Some(t).
26: **upon** $\mathcal{A}$ calls `SetLocalOutcome`(*id*, *outcome*):
27: $\quad$ **if** $global_outcome$ = None **then**
28: $\quad\quad$ **return** *InvalidOperation*.
29: $\quad$ **else if** $local_outcome[id] \neq$ None **then**
30: $\quad\quad$ **return** *InvalidOperation*.
31: $\quad$ **else if** *outcome* is not valid or consistent with the global outcome **then**
32: $\quad\quad$ **return** *InvalidOperation*.
33: $\quad$ **else if** *outcome* is not consistent with the local outcome of some other process **then**
34: $\quad\quad$ **return** *InvalidOperation*.
35: $\quad$ **else**
36: $\quad\quad local_outcome[id] \leftarrow$ Some$(outcome)$.

If D(*choice*) contains `Committed`(v) or `MaybeCommitted`(v) as a possible outcome, then we require there must be a valid proposal for v. Otherwise we say D(*choice*) is not allowed and `OpenGlobalOutcome`(*choice*) is rejected (line 22).

The Local Outcomes: After the global outcome is probabilistically determined, the adversary may call `SetLocalOutcome`($id, outcome$) to adaptively assign the local outcomes received by honest processes. However, they must be valid and consistent with the global outcome (line 28), and consistent with each other (line 30). To be "valid" means if the outcome is `Committed`(v) or `MaybeCommitted`(v), then there is a valid proposal for v. The consistency rules are:

- If the global outcome is `Committed`(v), or the local outcome of some process is `Committed`(v), then the local outcome of every process must be either `Committed`(v) or `MaybeCommitted`(v).
- If some process p_i receives `Committed`(v) or `MaybeCommitted`(v), and another process p_j receives `Committed`(v') or `MaybeCommitted`(v'), then $v = v'$.
- If the global outcome is `Committed`(v) but some process receives `MaybeCommitted`(v), then there exists a valid proposal for a value different from v. In other words, if honest processes only make validity votes for a single value v, and the global outcome is `Committed`(v), then all honest processes output `Committed`(v).

The point of GCA is to allow reasoning about the outcomes observed by each process axiomatically through these consistency rules, which will greatly simplify the liveness proof of BA. Readers may find our characterization of GCA more complex than expected. Some increase in complexity is unavoidable when translating the very informal specifications in works like [25] to a formal transition system model. We weakened the binding property of [2] to being only *probabilistically binding*: the adversary calls `OpenGlobalOutcome`($choice$) to reveal the global outcome, and binding is in effect only if the global outcome is `Committed`(v). Of course, this implies the coin-flipping step is now part of GCA (line 24), in contrast to [2] which formulate GCA deterministically.

There are two considerations behind our formulation of GCA: (1) If we make GCA deterministic, then the BA loop needs to interact with both an infinite sequence of GCA instances and an infinite sequence of common coins. Since each common coin is tied to a particular instance of GCA, we feel it is natural to consider coin-flipping as part of GCA. (2) The deterministic binding property is only possible in the binary setting, while probabilistic binding can be implemented also for multi-valued agreement, which facilitates porting our work to multi-valued agreement algorithms later.

Encoding the GCA Functionality in SureDistrib. We now use GCA as an example to show how to encode an abstract functionality in SureDistrib. Our theory of functionality composition borrows ideas from game-based semantics [41]. We encode GCA as a PTS whose step relation δ_p is the union of three types of steps:

- **Opponent moves** are the steps where an honest process calls one of the APIs. The APIs of GCA are `AddValidVote`(v), `Propose`(v), and `Learn`().
- **Proponent moves** are the steps where the functionality returns some information to an honest process. The `AddValidVote`(v) and `Propose`(v) calls complete atomically and return no information, so they have no proponent moves. The only proponent move is to return the local outcome after the process calls `Learn`().
- **Local moves** are the other steps that are invisible to the honest processes in the overlay application. This includes the byzantine processes calling `AddValidVote`(v) and `Propose`(v), and the adversary calling `OpenGlobalOutcome`($choice$) and `SetLocalOutcome`($ID, outcome$).

This work does not depend on further technical details of [41], but the connection between I/O automata and game-based semantics is an interesting direction we plan to investigate further.

We say a step $(s, D) \in \delta_p$ is non-probabilistic if D has only one element, i.e. $D = [(s', 1)]$ for some s'. In SureDistrib we postulate that all opponent and proponent moves are non-probabilistic. Specifically, let Γ be the set of API calls an honest process may make, and Θ be the set of values the functionality may return to the caller. We postulate there is a function $\delta_O : S \times ID \times \Gamma \mapsto S$ that

specifies the non-probabilistic state transition when process p_i calls some API in Γ. We also postulate that all proponent moves are contained in a set $\delta_P \in \mathcal{P}(S \times (S \times ID \times \Theta))$. If $(s, (s', i, v)) \in \delta_P$, this means the abstract functionality at state s may return v to process p_i, and transition to state s'. The local steps are represented by a set $\delta_L \in \mathcal{P}(S \times \mathtt{dist}_S)$; if $(s, D) \in \delta_L$ then D is normalized. The probabilistic step relation δ_p is defined by $\delta_O, \delta_P, \delta_L$ (Fig. 4).

In the artifact we simply represent δ_O as functions on the abstract state, and δ_P, δ_L are represented as inductive relations. This completes the encoding of GCA (Alg. 2). The remaining questions are: How to define an implementation of GCA, and prove that it refines Alg. 2? How to specify an overlay application that runs atop GCA? How to compose such an application with an implementation of GCA? We resolve these questions in Sec. 3. In Sec. 4, we present a formally verified implementation of Alg. 2, and a BA algorithm built on top of it.

$$\frac{p_i \text{ honest} \quad q \in \Gamma}{(s, [(\delta_O(s, i, q), 1)]) \in \delta_p} \text{ Opponent}$$

$$\frac{(s, (s', i, v)) \in \delta_P}{(s, [(s', 1)]) \in \delta_p} \text{ Proponent}$$

$$\frac{(s, D) \in \delta_L}{(s, D) \in \delta_p} \text{ Local}$$

Fig. 4. Opponent, proponent, and local moves of an abstract functionality.

2.3 Specifying Liveness Properties of Agreement Functionalities

To make liveness guarantees available to overlays which have no knowledge about the implementation, we specify them as safety properties on *segmented traces* of the abstract functionality (Fig. 5). Implementations of the functionality must then prove that their segmented traces refine the segmented traces of the abstract functionality.

Segmented traces were originally introduced by Qiu et al. [54] for (partially) synchronous protocols. They are an intuitive way to formalize the notion of *communication steps* typically employed in distributed system reasoning. Let us first review how trace segmentation works for partially-synchronous systems. These protocols assume two parameters denoted by *GST* (the global synchronization time) and Δ, such that after *GST*, each sent message will be delivered within Δ.

In temporal logic, the delivery guarantee for message m can be expressed as a fairness assumption $\Box(m \in \mathcal{M}_{sent} \rightarrow \Diamond(m \in \mathcal{M}_{deliv}))$ where $\mathcal{M}_{sent}$ is the set of sent messages and $\mathcal{M}_{deliv}$ is the set of delivered messages (see Sec. 3.2). At any given moment, there are only a finite number of active fairness assumptions since $\mathcal{M}_{sent}$ is always a finite set. The informal notion of a "communication step" can be defined as a time period, during which all fairness assumptions that became active before the start of the period are fulfilled by the end of the period. Under this notion, the partial-synchrony assumption can be formalized as saying each period of Δ is a communication step. Hence Qiu et al. [54] represents an infinite timed trace as the limit of an infinite number of finite traces $\tau_0, \tau_1, \tau_2, \cdots$, each τ_{k+1} being τ_k extended with events over a new period of Δ. Under this representation, we can state liveness guarantees in terms of the communication steps needed to achieve the results, which closely corresponds with how the protocols are analyzed on paper.

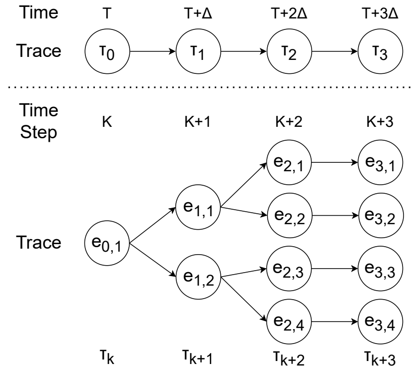

Fig. 5. Segmented traces under partial synchrony and asynchrony. Under partial synchrony we can represent an infinite timed trace with its snapshots at periods of Δ. This is the segmented trace idea of Qiu et al. [54]. Under asynchrony the trace is now a tree of probabilistic branches. With Def. 1, $\tau_k = [[e_{0,0}]]$, $\tau_{k+1} = [[e_{1,1}, e_{1,2}]]$, $\tau_{k+2} = [[e_{2,1}, e_{2,2}], [e_{2,3}, e_{2,4}]]$, and $\tau_{k+3} = [[e_{3,1}], [e_{3,2}], [e_{3,3}], [e_{3,4}]]$.

Asynchronous protocols assume no upper bound on network latency, so we cannot segment traces by periods of Δ. Still, the notion of a "communication step" is useful. Each step is no longer a

fixed period of time, but an arbitrarily long period needed to fulfill all the fairness assumptions: messages sent before the beginning of the step are delivered and processed by the end of the step.

An additional complication in our setting is that the trace is no longer linear but a probabilistic tree. Hence in a PTS trace $D \to_p^* D'$, when we consider a state $e' \in D'$ we also have to consider its origin $e \in D$. For example, if a message m was sent in state $e_1 \in D$ but not $e_2 \in D$ then m is eventually delivered only in states that branched out from e_1 but not e_2. Here Lemma 1 comes to the rescue. If $D \to_p^* D'$ then Lemma 1 says we can always split D' into $D'_1, \cdots, D'_n$, each representing the subtree of the trace that grew out of an element in D. We can then define liveness properties as safety properties that relate each element $e_i \in D$ to its subtree D'_i.

Definition 1. *A segmented trace of a PTS is a sequence $\tau_0, \tau_1, \tau_2, \cdots$ of values of type* $\texttt{list}(\texttt{dist}_S)$, *s.t.* $\texttt{length}(\tau_{k+1}) = \texttt{length}(\texttt{concat}(\tau_k))$, *and if e is the i-th element of* $\texttt{concat}(\tau_k)$, *D is the i-th element of τ_{k+1}, then $[e] \to_p^* D$. Here* $\texttt{concat}(\cdot)$ *is list concatenation, as each $D \in \texttt{dist}_S$ is itself a list.*

In other words, each $\texttt{concat}(\tau_k)$ represents the system state distribution at some instant, so that $\texttt{concat}(\tau_k) \to_p^* \texttt{concat}(\tau_{k+1})$, and the i-th element of τ_{k+1} represents the subtree that grew out of the i-th element of $\texttt{concat}(\tau_k)$. In Fig. 5, the subtree that grew out of $e_{0,1}$ contains $e_{1,1}, e_{1,2}$, and the subtree that grew out of $e_{1,1}$ contains $e_{2,1}, e_{2,2}$. We call the evolution from τ_k to τ_{k+1} a *time step*, which corresponds to the informal notion of an asynchronous communication step.

A Small Liveness Logic. To work effectively with probabilistic segmented traces, we define two operators which we denote by $\Diamond_n$ and $\Diamond_{n,p}$. Let $\mathcal{R}(s, s')$ be a binary relation on system states. We use $\Diamond_n(\mathcal{R})$ to denote that, given any k and any $e \in \texttt{concat}(\tau_k)$, let D be the sublist of τ_{k+n} that branched out from e, then every $e' \in D$ satisfies $(e.\text{s}, e'.\text{s}) \in \mathcal{R}$. In particular, the "always" operator $\Box(P)$ where P is a predicate can be defined as $\Diamond_0(s = s' \wedge P(s))$.

We use $\Diamond_{n,p}$ to denote that, given any k and any $e \in \texttt{concat}(\tau_k)$, let D be the sublist of τ_{k+n} that branched out from e, then there is a sublist D' of D, whose total probability is at least $p \cdot e.\text{p}$, and every $e' \in D'$ satisfies $(e.\text{s}, e'.\text{s}) \in \mathcal{R}$. It is similar to the $U \to_{t,p} U'$ operator in [45].

For example, "each message is eventually delivered" can be formally stated as $\forall m, \Diamond_1(m \in s.\mathcal{M}_{sent} \to m \in s'.\mathcal{M}_{deliv})$. This is a safety property over consecutive steps, rather than a liveness property. We proved a number of lemmas for manipulating and chaining $\Diamond_n$ and $\Diamond_{n,p}$ statements.

Almost-Sure Termination. With the above preparation we are now ready to state what is proved as "almost-sure termination" in this work. Let $\mathcal{R}(s, s')$ be a binary relation on system states.

Definition 2. *$\mathcal{R}$ almost surely holds, if there exist functions $f(n) : \mathbb{N} \mapsto [0, 1]$ and $t(n) : \mathbb{N} \mapsto \mathbb{N}$, s.t. $\Diamond_{t(n),f(n)}(\mathcal{R})$ holds for every n, and $\lim_{n\to\infty} f(n) = 1$.*

In Sec. 4, we present our modified version of Alg. 1. Let $\mathcal{R}(s, s')$ be the relation "if all honest processes have input a bit in s, then all honest processes have received an output in s'." Our formal liveness conclusion is that $\mathcal{R}$ holds almost surely. We prove this proposition by defining a state variable $\texttt{Hope}(s)$ which we call the "termination hope." Informally, it is a lower bound on the probability of terminiation within a given number of time steps. We prove that the expectation of $\texttt{Hope}(s)$ increases monotonically during execution. In the terminology of [62], $\texttt{Hope}(s)$ is a submartingale. We derive almost-sure termination from this fact. It is also possible to derive the expected number of time steps needed for termination. See Appendix A for details.

3 The SureDistrib Framework

3.1 Abstract Functionalities

We characterize an abstract functionality by a tuple $(S, s_0, \Gamma, \Theta, \delta_O, \delta_P, \delta_L)$, where S is the state space of the functionality, and s_0 is the initial state. The Γ set consists of the operations each honest

process may invoke on the functionality. We assume all functionalities in the system have the same set of participants, and for each functionality, the interfaces exposed to each honest participant are identical. The function $\delta_O : (S \times ID \times \Gamma) \mapsto S$ specifies the state change upon an honest process invoking an operation. Operations may be invoked at any given time. However, sometimes the invocation will be rejected (such as honest processes proposing two different values in an instance of agreement, line 14 of Alg. 2). Such invocations result in no change to the system state.

Some operations cannot be completed atomically. For example, the Learn() operation of GCA needs to wait until an outcome is actually available. Such operations are modeled by maintaining a list of processes currently waiting upon the call. Invoking another Learn() operation when the process is already waiting upon one is rejected as invalid operation.

The set Θ contains all values the functionality may return to the caller. Such return steps are specified by the set $\delta_P \in \mathcal{P}(S \times (S \times ID \times \Theta))$. Steps that are neither invocations nor returns are called local steps, and are specified by $\delta_L \in \mathcal{P}(S \times \mathtt{dist}_S)$. The state transitions of the functionality are thus defined by $\delta_O, \delta_P, \delta_L$, as shown in Fig. 4.

Functionalities Used in This Work. In this work we work with three concrete functionalities, which are common coins, graded crusader agreement (GCA), and binary agreement (BA). We already introduced GCA in Sec. 2. Here we define common coins and BA. The common coin functionality is defined in Alg. 3. Each process may call OpenCoin() to request revealing the coin. After $2f+1$ processes have called OpenCoin(), the adversary may call RevealCoin() to sample the coin value from a fixed distribution D. In this work we only use binary common coins, which are common coins with domain $V = \{0, 1\}$, each with probability $1/2$. Common coins cannot be implemented under our system model, and in this work we treat them as black box primitives.

Algorithm 3 Abstract model for common coins

```
1:  Implementation-specific parameters:
2:    V, the domain of output values.
3:    D, a normalized distribution over V.
4:  State variables:
5:    open_votes : list ID, list of processes voted
    to reveal the coin value.
6:    value : option V, the value of the coin.
7:  upon p_id calls OpenCoin():
8:      open_votes ← id :: open_votes.
9:      Wait until value = Some(v).
10:     return v.
11: upon 𝒜 calls RevealCoin():
12:     if value ≠ None then
13:         return InvalidOperation.
14:     else if open_votes does not contain 2f + 1
    processes then
15:         return InvalidOperation.
16:     else
17:         v ←$ D.
18:         value ← Some(v).
```

Algorithm 4 Abstract model for binary agreement

```
1:  Notation: B = {0, 1}.
2:  State variables:
3:    proposals : list(ID×B), list of proposers and
    their proposals.
4:    outcome : option B, the outcome of this BA
    instance.
5:  upon p_id calls BAPropose(b):
6:      if p_id is honest and has already called
    BAPropose(v') for some v' then
7:          return InvalidOperation.
8:      proposals ← (id, b) :: proposals.
9:  upon p_id calls Learn():
10:     Wait until outcome = Some(b).
11:     Return b.
12: upon 𝒜 calls SetOutcome(b):
13:     if outcome ≠ None then
14:         return InvalidOperation.
15:     else if proposals does not contain proposals
    from 2f + 1 processes then
16:         return InvalidOperation.
17:     else if b was not proposed by some honest
    process then
18:         return InvalidOperation.
19:     else
20:         outcome ← Some(b).
```

The BA functionality is defined in Alg. 4. Each process may call BAPropose(b) to propose a bit b. Each honest process may only make one proposal. After $2f + 1$ processes have submitted proposals, the adversary may call SetOutcome(b) to set the agreement outcome. Subsequently, the honest processes will receive this outcome. Unlike the GCA functionality, the global outcome of BA is not chosen probabilistically. The adversary may set the global outcome to any value proposed by at least one honest process.

3.2 Overlays of Functionalities

We will implement binary GCA upon binary common coins, and BA upon GCA. To do so, we first need a formalism for specifying overlay applications atop underlay functionalities. Typically, in such applications each honest process will maintain some state variables, and exchange messages independent of the underlay. We use S_{local} to denote the state space of each honest process. The initial state of each honest process is s_0^{local}. We use $\mathcal{M}$ to denote the set of messages that may be exchanged between processes. Each message m has a sender, denoted by $m.sender$, and we assume each process can only fill this field with its own ID. We represent the interface of the application by the pair (Γ', Θ') to distinguish it from the underlay interface (Γ, Θ).

Several kinds of events may occur in the application system:

- An honest process may accept an invocation, or return some value to the caller. We specify these events with $\delta'_O : (ID \times S_{local}) \times \Gamma' \mapsto S_{local}$ and $\delta'_P \in \mathcal{P}((ID \times S_{local}) \times (S_{local} \times \Theta'))$. The prime distinguishes these entities from those of the underlay. In general, we do not assume the ID of a process is stored inside the process local state. Therefore, when specifying the behaviors of honest processes we provide its ID as an argument.
- An honest process may send a message, or invoke an operation of the underlay. We specify these events with $\delta'_{L,send} \in \mathcal{P}((ID \times S_{local}) \times (S_{local} \times \mathcal{M}))$ and $\delta'_{L,call} \in \mathcal{P}((ID \times S_{local}) \times (S_{local} \times \Gamma))$. We require that if $((i, s), (s', m)) \in \delta'_{L,send}$, then $m.sender = i$.
- An honest process may receive a message, or some value from the underlay. We specify these events with $\delta'_{L,recv} : (ID \times S_{local} \times \mathcal{M}) \mapsto S_{local}$ and $\delta'_{L,return} : (ID \times S_{local} \times \Theta) \mapsto S_{local}$.
- An honest process may make a probabilistic or non-probabilistic state transition. These events are specified with $\delta'_{L,prob} \in \mathcal{P}((ID \times S_{local}) \times \mathtt{dist}_{S_{local}})$.
- A byzantine process may inject a message into the network, or the underlay may make a local move (which includes byzantine processes calling interfaces of the underlay). Both kinds of events do not have an immediate effect on the honest processes of the overlay.

We maintain some ghost variables regarding message broadcast and delivery. We maintain a set $\mathcal{M}_{sent} \subseteq \mathcal{M}$ for the set of all messages ever sent into the network. We also maintain a set $\mathcal{M}_{deliv} \in \mathcal{P}(ID \times \mathcal{M})$ of message delivery records. We use S_{msg} to denote the record type that carries $\mathcal{M}_{sent}$ and $\mathcal{M}_{deliv}$. Thus the complete state of the system is a value of type $\mathtt{FinMap}(ID \mapsto S_{local}) \times S_{msg} \times S_{underlay}$. In Fig. 6, we show the semantics of each possible step.

3.3 Refinement

What does it mean that an application built on top of a common coin implements GCA? A necessary condition is the application $\mathcal{G}$ provides the same interface (Γ, Θ) as the abstract model $\mathcal{F}$ of GCA, but this is clearly not sufficient. We also need to prove that the behavior of $\mathcal{G}$ simulates the behavior of $\mathcal{F}$. Formally, this is achieved via a *refinement* proof, which consists of a refinement relation $\mathcal{R}$, and proofs for the following propositions:

- The initial states of $\mathcal{F}, \mathcal{G}$ are related by $\mathcal{R}$.
- Each opponent move of $\mathcal{G}$ (Opponent in Fig. 6) refines the corresponding opponent move of $\mathcal{F}$ (the same process invoking the same operation).
- For each proponent move of $\mathcal{G}$ (Proponent in Fig. 6), there is a proponent move of $\mathcal{F}$ that returns the same value to the same process, and the end states of $\mathcal{F}, \mathcal{G}$ are related by $\mathcal{R}$.
- For each local move of $\mathcal{G}$ (all steps other than Opponent and Proponent in Fig. 6), there exist a sequence of zero or more local moves of $\mathcal{F}$ and split steps (Fig. 3), such that the final state distributions of $\mathcal{F}, \mathcal{G}$ have the same number of elements, and each pair of corresponding elements have the same probability value, and the states are related by $\mathcal{R}$.

$$\frac{p_i \text{ honest} \quad s_{local}[i] = s \quad q \in \Gamma'}{(s_{local}, s_{msg}, s_{und}) \rightarrow_p [((s_{local}[i \leftarrow \delta'_O(i,s,q)], s_{msg}, s_{und}), 1)]} \text{ Opponent}$$

$$\frac{p_i \text{ honest} \quad s_{local}[i] = s \quad ((i,s),(s',v)) \in \delta'_P}{(s_{local}, s_{msg}, s_{und}) \rightarrow_p [((s_{local}[i \leftarrow s'], s_{msg}, s_{und}), 1)]} \text{ Proponent}$$

$$\frac{p_i \text{ honest} \quad s_{local}[i] = s \quad ((i,s),(s',m)) \in \delta'_{L,send}}{(s_{local}, s_{msg}, s_{und}) \rightarrow_p [((s_{local}[i \leftarrow s'], s_{msg}[\mathcal{M}_{sent} \mathrel{+}= m], s_{und}), 1)]} \text{ Send}$$

$$\frac{p_i \text{ honest} \quad s_{local}[i] = s \quad ((i,s),(s',q)) \in \delta'_{L,call}}{(s_{local}, s_{msg}, s_{und}) \rightarrow_p [((s_{local}[i \leftarrow s'], s_{msg}, \delta_O(s_{und}, i, q)), 1)]} \text{ Call}$$

$$\frac{p_i \text{ honest} \quad s_{local}[i] = s \quad m \in \mathcal{M}_{sent}}{(s_{local}, s_{msg}, s_{und}) \rightarrow_p [((s_{local}[i \leftarrow \delta'_{L,recv}(i,s,m)], s_{msg}[\mathcal{M}_{deliv} \mathrel{+}= (i,m)], s_{und}), 1)]} \text{ Deliver}$$

$$\frac{p_i \text{ honest} \quad s_{local}[i] = s \quad (s_{und}, (s', i, v)) \in \delta_P}{(s_{local}, s_{msg}, s_{und}) \rightarrow_p [((s_{local}[i \leftarrow \delta'_{L,return}(i,s,v)], s_{msg}, s'), 1)]} \text{ Return}$$

$$\frac{m \in \mathcal{M} \quad m.sender \text{ byzantine}}{(s_{local}, s_{msg}, s_{und}) \rightarrow_p [((s_{local}, s_{msg}[\mathcal{M}_{sent} \mathrel{+}= m], s_{und}), 1)]} \text{ Byzantine Send}$$

$$\frac{p_i \text{ honest} \quad s_{local}[i] = s \quad ((i,s), D) \in \delta'_{L,prob}}{(s_{local}, s_{msg}, s_{und}) \rightarrow_p \mathtt{map}(e \mapsto ((s_{local}[i \leftarrow e.\mathrm{s}], s_{msg}, s_{und}), e.\mathrm{p}))\ D} \text{ Overlay Local}$$

$$\frac{(s_{und}, D) \in \delta_L}{(s_{local}, s_{msg}, s_{und}) \rightarrow_p \mathtt{map}(e \mapsto ((s_{local}, s_{msg}, e.\mathrm{s}), e.\mathrm{p}))\ D} \text{ Underlay Local}$$

Fig. 6. Semantics of an overlay application. Here s_{local} is a finite map from process ID to the state of an honest process; s_{msg} consists of $\mathcal{M}_{sent}$ and $\mathcal{M}_{deliv}$, and s_{und} is the underlay state. The overlay defines the δ' entities, while the underlay defines the δ entities. In this figure $s \rightarrow_p D$ means $[(s,1)] \rightarrow_p D$.

Our notion of refinement is closely related to the *strong observational refinement* of probabilistic programs [5]. It is also related to simulation-based security proofs [44] used in cryptography: the refinement proof serves as the simulator that translates local moves of the implementation to local moves of the abstract model. However, in our setting the adversary always has full knowledge of the system, so we do not need to prove indistinguishability of states.

3.4 Parallel Composition

The BA algorithm depends not on a single instance of GCA, but rather a countably infinite sequence of GCA instances. We now define an operator $\bigotimes_{r=0}^{\infty}$ that represents the parallel composition of an infinite number of identical functionalities. The definition is straightforward. We use $\mathtt{FinMap}(S)$ to represent the state space of the composed system, where S is the state space of the original system. If the state of instance r is not defined in the finite map, we take its state to be s_0, the initial state of the original system. We replace the interface of the system (Γ, Θ) with $(\mathbb{N} \times \Gamma, \mathbb{N} \times \Theta)$. The transition relations $\delta_O, \delta_P, \delta_L$ are similarly replaced with a countably infinite number of copies.

The parallel composition of overlay applications can be similarly defined. We replace the underlay functionality $\mathcal{F}$ with $\bigotimes_{r=0}^{\infty} \mathcal{F}$. The local state of each process is now $\mathtt{FinMap}(S_{local})$. If the state of instance r is undefined, it is taken to be s_0^{local}. The interface (Γ', Θ'), the message space $\mathcal{M}$, and the transition relations δ' are replaced by an infinite number of copies.

Lemma 2. *If an application $\mathcal{G}$ refines a functionality $\mathcal{F}$, then $\bigotimes_{r=0}^{\infty} \mathcal{G}$ refines $\bigotimes_{r=0}^{\infty} \mathcal{F}$.*

This is proved by projecting the state of the composed system to each individual instance. Each step affects only a single instance, and is a split event from the perspective of all other instances.

3.5 Layered Composition

Given an application $\mathcal{G}^{(1)}$ that implements $\mathcal{F}_2$ with underlay $\mathcal{F}_1$, and another application $\mathcal{G}^{(2)}$ that implements $\mathcal{F}_3$ with underlay $\mathcal{F}_2$, we can define a composed application $\mathcal{G}^{(1)} \odot \mathcal{G}^{(2)}$ that implements $\mathcal{F}_3$ with underlay $\mathcal{F}_1$. The state space of each honest process is $S^{(1)}_{local} \times S^{(2)}_{local}$, and the message space is the disjoint union of $\mathcal{M}^{(1)}$ and $\mathcal{M}^{(2)}$. The opponent and proponent steps are inherited from $\mathcal{G}^{(2)}$. The local steps are the union of the local steps of $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$. Each time $\mathcal{G}^{(2)}$ makes a call on $\mathcal{F}_2$, it is processed by the δ'_O handler supplied by $\mathcal{G}^{(1)}$. When $\mathcal{G}^{(1)}$ returns some value, it is processed by the $\delta'_{L,return}$ handler supplied by $\mathcal{G}^{(2)}$.

Lemma 3. *If $\mathcal{G}^{(1)}$ implements $\mathcal{F}_2$ with $\mathcal{F}_1$, and $\mathcal{G}^{(2)}$ implements $\mathcal{F}_3$ with $\mathcal{F}_2$, then $\mathcal{G}^{(1)} \odot \mathcal{G}^{(2)}$ implements $\mathcal{F}_3$ with $\mathcal{F}_1$.*

Each local move of $\mathcal{G}^{(1)} \odot \mathcal{G}^{(2)}$ affects either the local state of $\mathcal{G}^{(2)}$, the local state of $\mathcal{G}^{(1)}$, or the state of $\mathcal{F}_3$. In the first case, the refinement proof of $\mathcal{G}^{(2)}$ translates it into a sequence of local moves of $\mathcal{F}_3$. In the latter two cases, the refinement proof of $\mathcal{G}^{(1)}$ translates it into a sequence of local moves of $\mathcal{F}_2$, and then the refinement proof of $\mathcal{G}^{(2)}$ translates it into local moves of $\mathcal{F}_3$.

3.6 Compositional Liveness Proofs

Under SureDistrib liveness properties are stated as safety properties on segmented traces. Therefore they can be proved following the same methodology for composing safety proofs.

Let $\mathcal{G}$ be an application that refines a functionality $\mathcal{F}$ with refinement relation $\mathcal{R}$. We say a distribution $D \in \mathtt{dist}_{S_\mathcal{G}}$ refines a distribution $D' \in \mathtt{dist}_{S_\mathcal{F}}$, if D, D' have the same number of elements, and corresponding elements in D, D' have equal probability values, and the states are related by $\mathcal{R}$. Then from each segmented trace $\tau_0, \tau_1, \tau_2, \cdots$ of $\mathcal{G}$ we can construct a segmented trace $\tau'_0, \tau'_1, \tau'_2, \cdots$ of $\mathcal{F}$, such that τ_k and τ'_k have the same length, and each element $D \in \tau_k$ refines the corresponding element $D' \in \tau'_k$. We say $\{\tau'_k\}$ is the trace $\{\tau_k\}$ lifted to $\mathcal{F}$.

If P is a liveness property on $\mathcal{F}$ (a statement of $\Diamond_n$ or $\Diamond_{n,p}$), and Q is a liveness property on $\mathcal{G}$, we say Q refines P if every segmented trace of $\mathcal{G}$ that satisfies Q will also satisfy P when the trace is lifted to $\mathcal{F}$. If $\mathcal{G}'$ is an application that depends on $\mathcal{F}$, and liveness of $\mathcal{G}'$ can be proved assuming $\mathcal{F}$ satisfies P, then liveness of the composite system $\mathcal{G} \odot \mathcal{G}'$ also holds, provided the liveness assumption Q of $\mathcal{G}$ is respected. Therefore, it is sufficient to prove that each individual application satisfies the expected liveness properties of the overlay functionality, assuming liveness of the underlay functionality.

4 A Compositional Implementation of an Asynchronous Binary Agreement Algorithm

We now describe our verified implementation of BA. As shown in Fig. 1, the algorithm has two layers: `MSFA` which implements binary GCA using common coin, and `BAImpl` which implements BA using $\bigotimes_{r=0}^{\infty}$ `GCA`. The whole implementation is thus $\left(\bigotimes_{r=0}^{\infty} \mathtt{MSFA}\right) \odot \mathtt{BAImpl}$.

4.1 The Binary GCA Implementation

As explained in Sec. 2.1, the BA algorithm listed in Alg. 1 does not achieve liveness, because each round of Alg. 1 fails the binding property of Abraham et al. [2]. MacBrough [48] suggested adding a phase where each process broadcasts the values for which valid `AUX` messages have been observed. The message for this phase is called `CONF` in [48], but we call it `ASET` which stands for "set of `AUX`."

Algorithm 5 Binary GCA Implementation (code for each honest process)

```
1: State variable: local_outcome initialized to None.
2: upon p_id calls AddValidVote(b):
3:     Broadcast ⟨BVAL, id, b⟩.
4: upon p_id calls Propose(b):
5:     Broadcast ⟨AUX, id, b⟩ if this process did not broadcast any AUX message previously.
6: upon p_id calls Learn():
7:     Wait until local_outcome = Some(t), return t.
8: upon receiving any message:
9:     Forward the message to other processes.
10: upon receiving ⟨AUX, id′, b⟩ for a particular (id′, b), and the process has previously received ⟨BVAL, _, b⟩ from 2f + 1 processes:
11:     Mark ⟨AUX, id′, b⟩ as valid.
12: upon observing valid ⟨AUX, _, _⟩ from 2f + 1 processes:
13:     q_0 ← whether valid ⟨AUX, _, 0⟩ observed.
14:     q_1 ← whether valid ⟨AUX, _, 1⟩ observed.
15:     Broadcast ⟨ASET, id, q_0, q_1⟩ if this process did not broadcast any ASET message previously.
16: upon receiving ⟨ASET, id′, q_0, q_1⟩:
17:     Discard the message if q_0 = q_1 = false.
18:     Mark the message as valid when the two conditions are both met:
19:     1) q_0 = false or a valid ⟨AUX, _, 0⟩ observed.
20:     2) q_1 = false or a valid ⟨AUX, _, 1⟩ observed.
21: upon observing valid ⟨ASET, _, _, _⟩ from 2f + 1 processes and this process has sent an ASET message:
22:     Call OpenCoin().
23: upon receiving the value of the common coin:
24:     B ← value of the common coin.
25:     vals ← {b | valid ⟨AUX, _, b⟩ received}.
26:     if vals = {B} then
27:         local_outcome ← Some(Committed(B)).
28:     else if vals = {0, 1} then
29:         local_outcome ← Some(MaybeCommitted(B)).
30:     else
31:         local_outcome ← Some(NotCommitted).
```

Our implementation of binary GCA is shown in Alg. 5, with changes from Alg. 1 marked in blue. Alg. 5 only specifies the behavior of each honest process. The refinement proof will simulate the adversary events like OpenGlobalOutcome(*choice*) from network-level adversary actions. Since Alg. 5 represents only a single round of BA, the messages do not have the round number field. We assume all messages are forwarded between honest processes (lines 8–9), which makes the liveness proof easier. We ignore the performance impact of this assumption. Upon observing $2f + 1$ valid AUX messages, instead of voting to open the common coin immediately, we broadcast an ASET message that indicates whether the process has observed valid AUX messages for each value (lines 12–15). The common coin is revealed only after an honest process has observed $2f + 1$ valid ASET messages (lines 21–22). Also, each honest process broadcasts at most one ASET message (line 15).

The Refinement Proof. When a byzantine process sends ⟨BVAL, _, b⟩, it is interpreted as calling AddValidVote(b), and sending ⟨AUX, _, b⟩ is interpreted as Propose(b). The key part of the refinement proof is to determine the global outcome when the common coin is revealed, and show that the local outcomes received by each process (lines 23–31) are consistent with this global outcome. Our probabilistic binding requirement says that the global outcome must become Committed(_) with non-zero probability. We first observe the following lemma:

Lemma 4. *By the moment the first honest process calls* OpenCoin() *(line 22), there exists a bit B, such that any honest process that calls* OpenCoin() *will have observed a valid* AUX *for B.*

Proof: The first honest process that calls OpenCoin() must have observed $2f + 1$ valid ASET messages. Among these messages, let T be the set of ASET messages from honest processes, and $|T| \geq f + 1$. Then every honest process must observe at least one member of T before calling OpenCoin(). If every $m \in T$ has $q_0 = true$ then every honest process must observe a valid AUX message for 0. Otherwise, some $m \in T$ has $q_0 = false$ and $q_1 = true$. Then there exist $2f + 1$ valid AUX messages for 1, and every honest process must observe at least one valid AUX message for 1. □

Based on Lemma 4, the refinement proof simulates OpenGlobalOutcome(*choice*) by following Alg. 6. When the adversary calls RevealCoin(), some honest process must have called OpenCoin(). We find a bit B that satisfies Lemma 4, and call OpenGlobalOutcome(B). The network state now

Algorithm 7 BA Implementation (code for each honest process)

1: State variable: *outcome* initialized to None.
2: **upon** p_{id} calls BAPropose(b):
3: Broadcast ⟨INPUT, $id, 0, b$⟩ if BAPropose(b') was never called for any b'.
4: **upon** p_{id} calls Learn():
5: Wait until *outcome* = Some(b), return b.
6: **upon** receiving ⟨INPUT, _, r, b⟩ for a particular (r, b) from $f + 1$ processes:
7: Broadcast ⟨INPUT, id, r, b⟩.
8: Call AddValidVote(b) on the r-th instance of GCA.
9: **upon** receiving ⟨INPUT, _, r, b⟩ for a particular (r, b) from $2f + 1$ processes:
10: Call Propose(b) on the r-th instance of GCA if this process did not propose any input in the r-th instance.
11: Call Learn() on the r-th instance of GCA.
12: **upon** receiving ⟨OUTPUT, _, b⟩ for a particular b from $f + 1$ processes:
13: Broadcast ⟨OUTPUT, _, b⟩.
14: **upon** receiving ⟨OUTPUT, _, b⟩ for a particular b from $2f + 1$ processes:
15: *outcome* ← Some(b); terminate execution.
16: **upon** receiving outcome of the r-th instance of GCA:
17: **if** outcome is Committed(b) **then**
18: Broadcast ⟨OUTPUT, id, b⟩.
19: Broadcast ⟨INPUT, $id, r + 1, b$⟩.
20: **else if** outcome is MaybeCommitted(b) **then**
21: Broadcast ⟨INPUT, $id, r + 1, b$⟩.
22: **else**
23: Broadcast ⟨INPUT, $id, r + 1, b$⟩ for every b upon observing ⟨INPUT, _, r, b⟩ from $f + 1$ processes.

splits into two states, where the coin value B' is 0 and 1 respectively. The abstract functionality state also splits into two states according to $D(B)$ (line 3). The network state where $B = B'$ refines the Committed(B) state, and the $B \neq B'$ state refines the NotCommitted state. Then it is easy to prove that the local outcomes (lines 23–31 of Alg. 5) are consistent with this global outcome. The global outcome becomes Committed(_) with probability 1/2.

Algorithm 6 Setting the Global Outcome

1: Implementation-provided parameters:
2: Choice set $\mathcal{I} = \{0, 1\}$.
3: Global outcome distribution $D(b) = [(\texttt{Committed}(b), 1/2), (\texttt{NotCommitted}, 1/2)]$.
4: **upon** $\mathcal{A}$ calls RevealCoin():
5: Find honest p_i that called OpenCoin().
6: Read p_i state to find bit B that satisfies Lemma 4.
7: Call OpenGlobalOutcome(B). (see Alg. 2)

The Liveness Proof. To guarantee progress of GCA, we require that: 1) every honest process eventually proposes a value; 2) after an honest process proposes a value b, every honest process makes a validity vote for b within one time step. Under these assumptions, after each honest process has proposed a value, every honest process will receive $2f + 1$ AUX messages within one time step. They also make validity votes for the proposed values. Therefore, they will mark these AUX messages as valid and broadcast an ASET message within two time steps. After any honest process broadcasts an ASET message, all honest processes will mark the message as valid in one time step, since we assume the BVAL and AUX messages are forwarded between processes. Hence, within three timesteps every honest process calls OpenCoin(). If the coin value is revealed within one time step, then every honest process returns an outcome within four time steps.

4.2 The BA Implementation

The BA algorithm is shown in Alg. 7. The INPUT messages are used to coordinate the honest processes to build validity votes and proposals. Each INPUT message has a round number r and a bit b. The OUTPUT messages represent the *termination gadget* [1]: they ensure each honest process can terminate without executing the GCA rounds indefinitely. These messages do not have the round number field. After each honest process has submitted an input (lines 2–3), there will be either at least $f + 1$ INPUT messages for 0, or at least $f + 1$ INPUT messages for 1 in round 0. Therefore, there is at least one value b for which every honest process will execute lines 7–8. When the honest processes observe the $2f + 1$ INPUT messages for b, they propose the value b (line 10). It is also easy to see that if any honest process executes line 10 to propose a value b, then all honest processes will

execute line 8 to build a validity vote for b in one time step. Therefore, by the progress guarantee of GCA in Sec. 4.1, we get the outcome of round 0 in six time steps. After each honest process receives the outcome of round r, it broadcasts at least one `INPUT` message for the next round (lines 16–23). The process repeats for each round until some honest process terminates (line 15). From this description we see that:

Lemma 5. *If no honest process terminates, then eventually a new round of GCA is completed.*

The Refinement Proof. The BA algorithm maintains the following invariant:

Lemma 6. *If the global outcome of the r-th instance of GCA is* `Committed`(b)*, or some honest process receives* `Committed`(b) *from the r-th instance, then in every round $r' > r$ all honest processes only broadcast* `INPUT` *messages for b.*

Proof: If in the r-th instance of GCA, the global outcome is `Committed`(b), or some honest process receives `Committed`(b), then by the consistency rules of GCA (see Sec. 2.2), every honest process receives either `Committed`(b) or `MaybeCommitted`(b). They will all execute line 19 or 21 to broadcast the initial `INPUT` messages in round $r + 1$. From this we see that all honest `INPUT` messages in round $r + 1$ are for b. It follows, the outcome received by any honest process in round $r + 1$ can only be `Committed`(b), `MaybeCommitted`(b), or `NotCommitted`. Thus all honest `INPUT` messages in round $r + 2$ must be for b as well. The higher rounds follow the same reasoning. □

Now an honest process broadcasts an `OUTPUT` message for b only when receiving a `Committed`(b) output from some instance of GCA (line 18), or upon observing $f + 1$ `OUTPUT` messages for b (line 13). By Lemma 6 it can be proved that all honest `OUTPUT` messages must be for the same value b. Therefore, at the moment the first honest process broadcasts $\langle$`OUTPUT`$, id, b\rangle$, we interpret it as the adversary calling `SetOutcome`(b). An honest process outputs b only upon receiving $2f + 1$ `OUTPUT` messages for b (line 15), at which point the global outcome must have been already set to b.

The Liveness Proof. Three liveness properties of Alg. 7 can be proved deterministically:

- If no honest process terminates, then eventually a new round of GCA is completed.
- If one honest process terminates (line 15), then all processes will terminate.
- If two rounds of GCA have global outcome `Committed`(b), then all processes terminate.

The first property is proved as Lemma 5 above. For the second property: if one honest process terminates by observing $2f + 1$ `OUTPUT` messages for the same bit b, then every honest process receives at least $f + 1$ `OUTPUT` messages for b in one time step. They will all broadcast `OUTPUT` messages for b (line 13), and everyone will terminate in two time steps.

For the third property: if there are two rounds $r < r'$ whose global outcomes are both `Committed`(b), then by Lemma 6, all honest `INPUT` messages in round r' are for b. Now by the consistency rules of GCA, every process will receive `Committed`(b) in round r'. They will all broadcast `OUTPUT` messages for b, and everyone will terminate in one time step. It follows that to prove almost-sure termination of BA, we only need to estimate the probability of getting two rounds with global outcome `Committed`(b), which is what we discuss next.

4.3 The Almost-Sure Termination Proof

To prove almost-sure termination of BA, we assume that every time the adversary sets the global outcome of an instance of GCA by calling `OpenGlobalOutcome`$(choice)$, there is a non-zero lower bound p_0 on the probability that the global outcome becomes `Committed(_)`. For Alg. 5, we proved that this probability is $1/2$. Now it is tempting to think that we can treat each instance of GCA as an independent event with probability p_0. This idea does not work because strictly speaking, the individual instances of GCA are not independent. The input to each round depends on the output

of the previous round, and the adversary can manipulate the outcome distribution adaptively. Thus the almost-sure termination proof is not so straightforward.

Our liveness proof is based on the following consideration: for a given state s of Alg. 7 and a positive integer N, what is the probability p that, starting from state s, eventually either some honest process terminates, or we get two rounds $r, r' < N$ with global outcome `Committed(_)`? We cannot compute the exact value of p, but we can derive a lower bound for p, which we call the "hope value" of s. Formally, we define two functions:

$$f_1(k) = \begin{cases} 0 & (k = 0) \\ p_0 + (1 - p_0) f_1(k-1) & (k > 0) \end{cases}, \quad f_2(k) = \begin{cases} 0 & (k = 0) \\ p_0 \cdot f_1(k-1) + (1 - p_0) f_2(k-1) & (k > 0) \end{cases}.$$

Thus $f_1(k)$ is a lower bound on the probability that within k consecutive rounds, at least *one* round gets the global outcome `Committed(_)`, and $f_2(k)$ is a lower bound on the probability that within k consecutive rounds, at least *two* rounds get the global outcome `Committed(_)`. Now the hope value of s, denoted by $\mathsf{Hope}(s)$, is defined in Alg. 8. If some honest processes have already terminated, or we already have two committed rounds, then we are certain that all processes will terminate, and in this case we return 1 (lines 6–9). In the second case, we have already completed N rounds, but we did not get two committed rounds. In this case there is no hope to terminate execution within the first N rounds, and we return 0 (lines 10–11). In the third case, we have not yet completed the first N rounds. We have $N - T(s)$ rounds remaining, and we need to get $2 - C(s)$ rounds committed within these rounds. The probability of achieving this is estimated with the f_1 and f_2 functions defined above (lines 12–15). We now make two crucial observations:

Algorithm 8 The Hope Value

```
1:  Notations:
2:     N: a fixed integer, the number of rounds to consider.
3:     T(s): the total number of rounds in s with their global outcomes determined.
4:     C(s): the total number of rounds in s with global outcome Committed(_).
5:  procedure HOPE(s)
6:     if at least one honest process terminated then
7:        return 1.
8:     else if C(s) ≥ 2 then
9:        return 1.
10:    else if T(s) ≥ N then
11:       return 0.
12:    else if C(s) = 1 then
13:       return f1(N − T(s)).
14:    else
15:       return f2(N − T(s)).
```

LEMMA 7. *Let s be a state of Alg. 7 where either some honest process has terminated, or $T(s) \geq N$. Then $\mathsf{Hope}(s) = 1$ if some honest process has terminated, or $C(s) \geq 2$, and $\mathsf{Hope}(s) = 0$ otherwise.*

Proof: The lemma follows directly from the definition of $\mathsf{Hope}(s)$. □

LEMMA 8. *If $[(s_{before}, 1)] \to_p D$ is some action step of Alg. 7, where s_{before} is the system state before the step, then $\mathbb{E}_D[\mathsf{Hope}] \geq \mathsf{Hope}(s_{before})$. ($\mathbb{E}_D[\cdot]$ defined in Sec. 2.2)*

Proof: Clearly there are only two events that affect the value of $\mathsf{Hope}(s)$: 1) an honest process terminates, or 2) the adversary opens the global outcome of a new round. In the first case, the hope value is 1 in the end state, but we always have $\mathsf{Hope}(s_{before}) \leq 1$, so the inequality holds. In the second case, $T(s)$ increases by 1 in each end state; $C(s)$ increases by 1 with probability at least p_0, and stays the same with probability at most $1 - p_0$. If $C(s_{before}) \geq 2$, then we have $C(s) \geq 2$ in every end state, so $\mathbb{E}_D[\mathsf{Hope}] = 1 \geq \mathsf{Hope}(s_{before})$. If $C(s_{before}) = 1$, then we have

$$\mathbb{E}_D[\mathsf{Hope}] \geq p_0 + (1 - p_0) f_1(N - T(s_{before}) - 1) = f_1(N - T(s_{before})) = \mathsf{Hope}(s_{before}).$$

Similarly if $C(s_{before}) = 0$ then we have

$$\begin{aligned} \mathbb{E}_D[\mathsf{Hope}] &\geq p_0 \cdot f_1(N - T(s_{before}) - 1) + (1 - p_0) f_2(N - T(s_{before}) - 1) \\ &= f_2(N - T(s_{before})) = \mathsf{Hope}(s_{before}). \end{aligned}$$

Thus the expectation of $\mathsf{Hope}(s)$ never decreases. □

A corollary of Lemma 8 is that if $D \to_p^* D'$ is a sequence of steps of Alg. 7, then $\mathbb{E}_D[\mathsf{Hope}] \le \mathbb{E}_{D'}[\mathsf{Hope}]$. Now starting from the initial state of Alg. 7, eventually we will reach a state distribution D where in every state $e \in D$ either some honest process has terminated, or $T(e.s) \ge N$. By Lemma 7, in this case $\mathbb{E}_D[\mathsf{Hope}]$ is the total probability in D that either some honest process has terminated, or $C(e.s) \ge 2$, and every process will terminate in two time steps. In the initial state s_0 of Alg. 7, we have $T(s_0) = C(s_0) = 0$. Therefore by Lemma 8, we have $\mathbb{E}_D[\mathsf{Hope}] \ge \mathsf{Hope}(s_0) = f_2(N)$.

A minor technical issue with the above argument is that: by Def. 2 we need to prove probabilistic termination starting from *any* state s in which all honest processes have input a bit, and in general we do not have $T(s) = C(s) = 0$. This can be circumvented by replacing N in Alg. 8 with $N + T(s)$.

By induction we can prove the following closed form expressions for $f_1(k), f_2(k)$:

$$f_1(k) = 1 - (1 - p_0)^k, \;\; f_2(k) = 1 - (1 - p_0)^{k-1}(1 + (k-1)p_0).$$

From this it is easy to see $\lim_{N\to\infty} f_2(N) \to 1$.

4.4 Proof Effort in Rocq

A full breakdown of our proof effort in Rocq is shown in Table 1. The proofs were developed by two persons over three months. The "probabilistic transition systems" part contains a complete library for defining a PTS and working with its segmented traces, including the liveness logic based on $\Diamond_n$ and $\Diamond_{n,p}$. The "network configurations" part formalizes the standard $3f + 1$ network configuration. We expect these parts can be reused for many probabilistic distributed algorithms. The remaining parts are specific to the BA algorithm formalized in this work, and can be reused by incorporating the algorithm into more complex systems, like the asynchronous consensus algorithm of Cachin et al. [12].

Table 1. Proof Effort Breakdown

Part	Proof Effort (Lines of Rocq)
Probabilistic Transition Systems	3005
Network Configurations	757
Public Coin (abstract model)	381
GCA	
Abstract model and its parallel composition	2276
Implementation	1731
Refinement	1283
Liveness	420
BA	
Abstract model	222
Implementation	1855
Refinement	412
Liveness	2539

5 Experimental Evaluation

Our Rocq specification is detailed enough such that we can extract the specification into executable OCaml code. Our extracted code is an event driven program where an incoming network message triggers corresponding actions to the message. For example, after receiving a vote message, the code checks whether a quorum of votes are received, and sends new messages if needed. Messages generated by the event handlers are sent using a trusted network shim layer. The shim layer uses OCaml's Unix socket libraries and no changes are made to the extracted core logic. The common coin is simulated with a fixed array of random bits that is read upon calling $\mathsf{OpenCoin}()$.

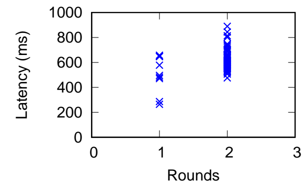

Fig. 7. Rounds and latencies to reach consensus.

We evaluated the executable code on a local cluster of four nodes, each with Intel Xeon Gold 6338 CPU, 128 GB memory, and 10 GigE NIC. We provided random binary inputs to each node and measured how many rounds and how much time it took to reach an agreement. Fig. 7 shows results from 140 runs; all runs completed at round 1 (9 out of 140) or 2 (131 out of 140) and the average latencies for each case were 496 and 630 ms, respectively. The experiment shows that despite probablistic guarantees, the protocol practically finishes within round 2.

6 Related Works

Asynchronous Consensus Protocols. Consensus under asynchrony is a problem with a very long history. Fischer et al. [28] proved that asynchronous consensus cannot be implemented deterministically, but Ben-Or [8] circumvented the impossibility by using randomization. Aguilera and Toueg [4] showed that Ben-Or's protocol maintains liveness under the public information model. Thus it is technically possible to analyze [8] under our framework. However, [8] has multiple probabilistic events in each round, so the formal proofs could be more difficult to develop.

Almost simultaneously with Ben-Or's result, Rabin [58] introduced the common coin primitive based on secret sharing, and described the first asynchronous consensus protocol that terminates with a constant number of expected rounds, but did not provide an efficient way to implement an infinite sequence of common coins. Cachin et al. [12] provided an implementation of common coins under the random oracle model [7] using deterministic threshold signatures, making Rabin [58]'s protocol practical.

Mostefaoui et al. [50] introduced an efficient common coin-based binary agreement algorithm, which was later incorporated into HoneyBadger [49]. However, it was soon found that [50] has a liveness bug against adaptive adversaries, which is the motivation for this work. Several recent works [31, 34, 64] have improved upon HoneyBadger. We plan to verify these schemes in the future.

There is also a line of work [20, 32, 63, 67] that seeks to combine the liveness guarantees of asynchronous protocols with the performance of partially synchronous protocols. While they represent a middle-ground between asynchrony and partial synchrony, they also introduce non-trivial liveness issues, such as reducing fallback to the asynchronous path [37].

Almost all known efficient common coin protocols are based on factoring or elliptic curve assumptions [26], which are not post-quantum secure. Designing and verifying a post-quantum asynchronous consensus protocol is an interesting challenge [29].

Verification of Distributed Algorithms. Many works have studied verifying safety and liveness properties of distributed algorithms. For example, Verdi [70], Disel [60], and Grove [61] all provided frameworks for safety verification. Qiu et al. [54], Zhao et al. [74] provide more extensive surveys of these results. Similar to our work, they decompose systems into modules using techniques like separation logic. These works can also handle safety of asynchronous consensus, since most of these protocols still provide deterministic safety. However, they cannot handle liveness properties.

Kwiatkowska et al. [43] presented mechanized proofs for a probabilistic consensus protocol. In this work, the probabilistic events are replaced with non-deterministic choices, so that only deterministic liveness properties are mechanized. The work used a model checker to compute the termination probability under a *fixed* number of processes. To model-check systems with a parameterized number of processes requires specialized model-checking techniques [39, 42], and these techniques have only recently been applied to asynchronous consensus [11, 30], with limitations discussed below. Our work gives mechanized proofs for the termination probability with any number of processes.

Bertrand et al. [10] reported model-checking liveness of a composite asynchronous consensus protocol. The liveness property proved in this work is deterministic, and the work established asynchronous liveness by introducing a non-standard fairness assumption.

Several works [18, 69] claimed verifying liveness of probabilistic consensus protocols. These works assumed that certain probabilistic event eventually occurs, and they derived deterministic liveness from this assumption. The probabilistic part is not verified.

Zhao et al. [74] introduced Bythos for verifying safety and liveness of composite asynchronous protocols using linear temporal logic (LTL). A similar work by Sun et al. [65] applied LTL to verify liveness of cluster managers. These works only handle deterministic properties, and so could

not handle asynchronous consensus. They do not use refinement to achieve layered composition. For example, Bythos could only handle *sequential* composition of primitives. As we have seen, establishing refinement between Alg. 5 and an abstract model of GCA is a crucial part of this work, and we think the proofs in this work would be much more difficult to construct without refinement.

Ticl [38] provides a more foundational treatment of temporal logic by introducing a computation model called *ICTrees*. The representation of a non-deterministic computation process as a tree structure is similar to our representation of probabilistic traces (Fig. 5). However, in our setting each branching corresponds to a probabilistic sampling, whereas in Ticl it is an adversarial non-deterministic choice. Ticl currently does not handle probabilistic liveness, but it seems promising to adapt its ideas to give a complete program logic for our computation model.

Two works [11, 30] achieved results very close to our work, both based on model-checking. Bertrand et al. [11] introduced a method for checking liveness of probabilistic threshold automata, and applied it to several algorithms including Ben-Or's protocol. Gao et al. [30] extended Bertrand et al. [11]'s approach to protocols based on common coins, and verified liveness of a modified version of Mostefaoui et al. [50]'s BA algorithm. Both works rely on a reduction to *round-rigid* adversaries that is only proved on paper. In effect, they only verified liveness of a single round of the protocol. Our work is the first foundational proof of the same result, and we also established almost-sure termination of the composite protocol.

A concurrent work of Enea et al. [24] verified almost-sure termination of a Ben-Or-like binary agreement algorithm using the super-martingale methodology of Chakarov and Sankaranarayanan [16]. Their proof requires defining both a *variant* function and a *super-martingale* on the system state, while our work requires only defining a submartingale for termination reasoning. Similar to our work, Enea et al. [24] decomposes BA into an outer loop and an inner GCA algorithm. However, Enea et al. [24] does not provide a way to compose probabilistic systems with cryptographic primitives, so they leave verifying the more practical common coin-based protocols to future work.

Our theory of functionality composition and refinement bears similarity to those for I/O automata [46]. The original theory was extended to probabilistic automata by Segala [59], Wu et al. [71], and then to *task-structured* automata [14] for modeling cryptography. Each component under our framework can be viewed as an I/O automaton or a composition of multiple I/O automata. The input events are receiving invocations from the overlay, messages from the network, or return values from the underlay. The output events are returning values to the overlay, broadcasting a message, or calling the underlay. The local moves are internal state transitions. The I/O automata formalism has been used to give on-paper analyses for several distributed algorithms [13, 53], but to our knowledge these works have never been mechanized. Indeed, the full I/O automata framework involves several more features that present additional challenges for verification. For example, modeling distributions with infinite support would require measure theory, but we assume all distributions are of finite support. As future work, we plan to investigate this connection and extend our mechanized framework to support general I/O automata.

Verification of Almost-Sure Termination. Proving a probabilistic program terminates almost surely is a challenge with a similarly long history [35], and martingales have proved to be a important tool in this area [16]. Some recent results are [27, 33, 47]. Most of these works target sequantial probabilistic programs, while probabilistic termination of concurrent programs remains largely unexplored, with the notable exception of Enea et al. [24] discussed above. Our work gives an example of encoding a martingale-based liveness proof of a distributed algorithm into a proof checker. The technical approach is slightly different in that we are using submartingales instead of ranking super-martingales like Enea et al. [24]. Our approach can be interpreted as a specialized application of the γ-scaled submartingale approach of Takisaka et al. [66].

Acknowledgments

We would like to thank the anonymous reviewers and the shepherd for their helpful feedback. This work is supported in part by a Sui Academic Research Award, by NSF grants 2019285, 2313433, 2442888, and by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590130. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

Data Availability Statement

The artifact accompanying this paper can be found at Qiu et al. [56].

References

[1] Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. 2022. Asynchronous Agreement Part 5: Binary Byzantine Agreement from a strong common coin. https://decentralizedthoughts.github.io/2022-04-05-aa-part-five-ABBA/. Accessed: November 2, 2025.

[2] Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. 2022. Efficient and Adaptively Secure Asynchronous Binary Agreement via Binding Crusader Agreement. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing* (Salerno, Italy) *(PODC'22)*. Association for Computing Machinery, New York, NY, USA, 381–391. doi:10.1145/3519270.3538426

[3] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 337–346. doi:10.1145/3293611.3331612

[4] Marcos K. Aguilera and Sam Toueg. 2012. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing* 25 (2012), 371–381. doi:10.1007/s00446-012-0162-z

[5] Hagit Attiya and Constantin Enea. 2019. Putting Strong Linearizability in Context: Preserving Hyperproperties in Programs That Use Concurrent Objects. In *33rd International Symposium on Distributed Computing (DISC 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 146)*, Jukka Suomela (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:17. doi:10.4230/LIPIcs.DISC.2019.2

[6] Hagit Attiya and Jennifer L. Welch. 2024. Multi-Valued Connected Consensus: A New Perspective on Crusader Agreement and Adopt-Commit. In *27th International Conference on Principles of Distributed Systems (OPODIS 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 286)*, Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:23. doi:10.4230/LIPIcs.OPODIS.2023.6

[7] Mihir Bellare and Phillip Rogaway. 1993. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (Fairfax, Virginia, USA) *(CCS '93)*. Association for Computing Machinery, New York, NY, USA, 62–73. doi:10.1145/168588.168596

[8] Michael Ben-Or. 1983. Another advantage of free choice (Extended Abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada) *(PODC '83)*. Association for Computing Machinery, New York, NY, USA, 27–30. doi:10.1145/800221.806707

[9] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. 1994. Asynchronous secure computations with optimal resilience (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, California, USA) *(PODC '94)*. Association for Computing Machinery, New York, NY, USA, 183–192. doi:10.1145/197917.198088

[10] Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazić, Pierre Tholoniat, and Josef Widder. 2022. Holistic Verification of Blockchain Consensus. In *36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:24. doi:10.4230/LIPIcs.DISC.2022.10

[11] Nathalie Bertrand, Igor Konnov, Marijana Lazić, and Josef Widder. 2021. Verification of randomized consensus algorithms under round-rigid adversaries. *International Journal on Software Tools for Technology Transfer* 23, 5 (2021), 797–821. doi:10.1007/s10009-020-00603-x

[12] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2000. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement using Cryptography. Cryptology ePrint Archive, Paper 2000/034. https://eprint.iacr.org/2000/034

[13] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. 2005. Using Probabilistic I/O Automata to Analyze an Oblivious Transfer Protocol. Cryptology ePrint Archive, Paper 2005/452. https://eprint.iacr.org/2005/452

[14] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. 2018. Task-structured probabilistic I/O automata. *J. Comput. System Sci.* 94 (2018), 63–97. doi:10.1016/j.jcss.2017.09.007 Journal of Computer and System Science: 50 years of celebration. In memory of Professor Edward Blum..

[15] Miguel Castro. 2001. *Practical Byzantine Fault Tolerance.* Ph. D. Dissertation. Massachusetts Institute of Technology. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/thesis-mcastro.pdf

[16] Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 511–526. doi:10.1007/978-3-642-39799-8_34

[17] Berk Cirisci, Constantin Enea, and Suha Orhun Mutluergil. 2023. Quorum Tree Abstractions of Consensus Protocols. In *Programming Languages and Systems*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 337–362. doi:10.1007/978-3-031-30044-8_13

[18] Karl Crary. 2021. Verifying the Hashgraph Consensus Algorithm. arXiv:2102.01167 [cs.LO] https://arxiv.org/abs/2102.01167

[19] F. Cristian, H. Aghili, R. Strong, and D. Dolev. 1995. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *Information and Computation* 118, 1 (1995), 158–179. doi:10.1006/inco.1995.1060

[20] Xiaohai Dai, Chaozheng Ding, Hai Jin, Julian Loss, and Ling Ren. 2024. Ipotane: Balancing the Good and Bad Cases of Asynchronous BFT. Cryptology ePrint Archive, Paper 2024/653. https://eprint.iacr.org/2024/653

[21] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 34–50. doi:10.1145/3492321.3519594

[22] Giovanni Deligios, Martin Hirt, and Chen-Da Liu-Zhang. 2021. Round-Efficient Byzantine Agreement and Multi-party Computation with Asynchronous Fallback. In *Theory of Cryptography: 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8–11, 2021, Proceedings, Part I* (Raleigh, NC, USA). Springer-Verlag, Berlin, Heidelberg, 623–653. doi:10.1007/978-3-030-90459-3_21

[23] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *Journal of the ACM* 35, 2 (April 1988), 288–323. doi:10.1145/42282.42283

[24] Constantin Enea, Rupak Majumdar, Harshit Jitendra Motwani, and V. R. Sathiyanarayana. 2026. Verifying Almost-Sure Termination for Randomized Distributed Algorithms. *Proc. ACM Program. Lang.* 10, POPL, Article 49 (Jan. 2026), 30 pages. doi:10.1145/3776691

[25] Paul Feldman and Silvio Micali. 1988. Optimal algorithms for Byzantine agreement. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing* (Chicago, Illinois, USA) *(STOC '88)*. Association for Computing Machinery, New York, NY, USA, 148–161. doi:10.1145/62212.62225

[26] Hanwen Feng and Qiang Tang. 2025. Asymptotically Optimal Adaptive Asynchronous Common Coin and DKG with Silent Setup. In *Advances in Cryptology – CRYPTO 2025*, Yael Tauman Kalai and Seny F. Kamara (Eds.). Springer Nature Switzerland, Cham, 642–675. doi:10.1007/978-3-032-01881-6_20

[27] Shenghua Feng, Mingshuai Chen, Han Su, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Naijun Zhan. 2023. Lower Bounds for Possibly Divergent Probabilistic Programs. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 99 (April 2023), 31 pages. doi:10.1145/3586051

[28] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (April 1985), 374–382. doi:10.1145/3149.214121

[29] Bryan Ford, Philipp Jovanovic, and Ewa Syta. 2020. Que Sera Consensus: Simple Asynchronous Agreement with Private Coins and Threshold Logical Clocks. *CoRR* abs/2003.02291 (2020). arXiv:2003.02291 https://arxiv.org/abs/2003.02291

[30] Song Gao, Bohua Zhan, Zhilin Wu, and Lijun Zhang. 2024. Verifying Randomized Consensus Protocols with Common Coins . In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, Los Alamitos, CA, USA, 403–415. doi:10.1109/DSN58291.2024.00047

[31] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Dumbo-NG: Fast Asynchronous BFT Consensus with Throughput-Oblivious Latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1187–1201. doi:10.1145/3548606.3559379

[32] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Financial Cryptography and Data Security*, Ittay Eyal and Juan Garay (Eds.). Springer International Publishing, Cham, 296–315. doi:10.1007/978-3-031-18283-9_14

[33] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Almost-Sure Termination by Guarded Refinement. *Proc. ACM Program. Lang.* 8, ICFP, Article 243 (Aug. 2024), 31 pages. doi:10.1145/3674632

[34] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Faster Asynchronous BFT Protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 803–818. doi:10.1145/3372297.3417262

[35] Sergiu Hart, Micha Sharir, and Amir Pnueli. 1983. Termination of Probabilistic Concurrent Program. *ACM Trans. Program. Lang. Syst.* 5, 3 (July 1983), 356–380. doi:10.1145/2166.357214

[36] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/2815400.2815428

[37] Rongji Huang, Xiangzhe Wang, Xiaofeng Yan, Lei Fan, Guangtao Xue, and Shengyun Liu. 2025. Chitu: avoiding unnecessary fallback in Byzantine consensus. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '25)*. USENIX Association, USA, Article 54, 20 pages.

[38] Eleftherios Ioannidis, Yannick Zakowski, Steve Zdancewic, and Sebastian Angel. 2025. Structural Temporal Logic for Mechanized Program Verification. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 313 (Oct. 2025), 28 pages. doi:10.1145/3763091

[39] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. 2013. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *2013 Formal Methods in Computer-Aided Design*. 201–209. doi:10.1109/FMCAD.2013.6679411

[40] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC'21)*. Association for Computing Machinery, New York, NY, USA, 165–175. doi:10.1145/3465084.3467905

[41] Jérémie Koenig and Zhong Shao. 2020. Refinement-Based Game Semantics for Certified Abstraction Layers. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 633–647. doi:10.1145/3373718.3394799

[42] Igor Konnov, Marijana Lazić, Ilina Stoilkovska, and Josef Widder. 2023. Survey on Parameterized Verification with Threshold Automata and the Byzantine Model Checker. *Logical Methods in Computer Science* Volume 19, Issue 1 (Jan. 2023). doi:10.46298/lmcs-19(1:5)2023

[43] Marta Kwiatkowska, Gethin Norman, and Roberto Segala. 2001. Automated Verification of a Randomized Distributed Consensus Protocol Using Cadence SMV and PRISM?. In *Computer Aided Verification*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–206. doi:10.1007/3-540-44585-4_17

[44] Yehuda Lindell. 2017. *How to Simulate It – A Tutorial on the Simulation Proof Technique*. Springer International Publishing, Cham, 277–346. doi:10.1007/978-3-319-57048-8_6

[45] Nancy Lynch, Isaac Saias, and Roberto Segala. 1994. Proving time bounds for randomized distributed algorithms. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing* (Los Angeles, California, USA) *(PODC '94)*. Association for Computing Machinery, New York, NY, USA, 314–323. doi:10.1145/197917.198117

[46] Nancy A. Lynch and Mark R. Tuttle. 1987. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, Fred B. Schneider (Ed.). ACM, 137–151. doi:10.1145/41840.41852

[47] Rupak Majumdar and V. R. Sathiyanarayana. 2024. Positive Almost-Sure Termination: Complexity and Proof Rules. *Proc. ACM Program. Lang.* 8, POPL, Article 37 (Jan. 2024), 29 pages. doi:10.1145/3632879

[48] Andrew Miller. 2018. Bug in ABA protocol's use of Common Coin. https://github.com/amiller/HoneyBadgerBFT/issues/59. Accessed: November 2, 2025.

[49] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. doi:10.1145/2976749.2978399

[50] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. 2014. Signature-free asynchronous byzantine consensus with t < n/3 and o(n2) messages. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing* (Paris, France) *(PODC '14)*. Association for Computing Machinery, New York, NY, USA, 2–9. doi:10.1145/2611462.2611468

[51] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf

[52] Natalie Neamtu, Haobin Ni, and Robbert Van Renesse. 2023. Trees and Turtles: Modular Abstractions for State Machine Replication Protocols. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data* (Rome, Italy) *(PaPoC '23)*. Association for Computing Machinery, New York, NY, USA, 9–15. doi:10.1145/3578358.3592148

[53] Anna Pogosyants, Roberto Segala, and Nancy Lynch. 2000. Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. *Distributed Computing* 13 (2000), 155–186. Issue 3. doi:10.1007/PL00008917

[54] Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, Wolf Honoré, and Zhong Shao. 2024. LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs. *Proc. ACM Program. Lang.* 8, PLDI, Article 193 (June 2024), 25 pages. doi:10.1145/3656423

[55] Longfei Qiu, Jingqi Xiao, Ji-Yong Shin, and Zhong Shao. 2025. LiDO-DAG: A Framework for Verifying Safety and Liveness of DAG-Based Consensus Protocols. *Proc. ACM Program. Lang.* 9, PLDI, Article 203 (June 2025), 25 pages. doi:10.1145/3729306

[56] Longfei Qiu, Jingqi Xiao, Ji Yong Shin, and Zhong Shao. 2026. *Artifact For PLDI 2026 Paper #304 SureDistrib: Verifying Almost-sure Termination of Composite Asynchronous Byzantine Protocols.* doi:10.5281/zenodo.19616610

[57] Longfei Qiu, Jingqi Xiao, Ji-Yong Shin, and Zhong Shao. 2026. *SureDistrib: Verifying Almost-Sure Termination of Composite Asynchronous Byzantine Protocols.* Technical Report YALEU/DCS/TR-1576. Yale Univ. https://flint.cs.yale.edu/publications/suredistrib.html

[58] Michael O. Rabin. 1983. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983).* 403–409. doi:10.1109/SFCS.1983.48

[59] Roberto Segala. 1996. *Modeling and verification of randomized distributed real-time systems.* Ph. D. Dissertation. USA. https://dspace.mit.edu/handle/1721.1/36560

[60] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (dec 2017), 30 pages. doi:10.1145/3158116

[61] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: A Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23).* Association for Computing Machinery, New York, NY, USA, 113–129. doi:10.1145/3600006.3613172

[62] Albert N. Shiryaev. 2019. *Probability-2.* Springer, New York, NY. doi:10.1007/978-0-387-72208-5

[63] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22).* Association for Computing Machinery, New York, NY, USA, 2705–2718. doi:10.1145/3548606.3559361

[64] Xiao Sui, Xin Wang, and Sisi Duan. 2025. Signature-Free Atomic Broadcast with Optimal $O(n^2)$ Messages and $O(1)$ Expected Time. In *2025 IEEE Symposium on Security and Privacy (SP).* 1547–1565. doi:10.1109/SP61157.2025.00244

[65] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: verifying liveness of cluster management controllers. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24).* USENIX Association, USA, Article 35, 18 pages.

[66] Toru Takisaka, Yuichiro Oyabu, Natsuki Urabe, and Ichiro Hasuo. 2021. Ranking and Repulsing Supermartingales for Reachability in Randomized Programs. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 5 (June 2021), 46 pages. doi:10.1145/3450967

[67] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galinanes, and Bryan Ford. 2023. QuePaxa: Escaping the tyranny of timeouts in consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23).* Association for Computing Machinery, New York, NY, USA, 281–297. doi:10.1145/3600006.3613150

[68] The Rocq Development Team. 2025. *The Rocq Prover.* doi:10.5281/zenodo.15149629

[69] Søren Eller Thomsen and Bas Spitters. 2021. Formalizing Nakamoto-Style Proof of Stake. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF).* 1–15. doi:10.1109/CSF51468.2021.00042

[70] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. *SIGPLAN Not.* 50, 6 (jun 2015), 357–368. doi:10.1145/2813885.2737958

[71] Sue-hwey Wu, Scott A. Smolka, and Eugene W. Stark. 1997. Composition and behaviors of probabilistic I/O automata. *Theoretical Computer Science* 176, 1 (1997), 1–38. doi:10.1016/S0304-3975(97)00056-X

[72] xygdys. 2022. A Bug in the Conf Phase of ABA protocol. https://github.com/initc3/HoneyBadgerBFT-Python/issues/57. Accessed: November 2, 2025.

[73] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19).* Association for Computing Machinery, New York, NY, USA, 347–356. doi:10.1145/3293611.3331591

[74] Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. 2024. Compositional Verification of Composite Byzantine Protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24).* Association for Computing Machinery, New York, NY, USA, 34–48. doi:10.1145/3658644.3690355

A More Details of the Probabilistic Temporal Logic

A.1 Inference Rules for the Temporal Logic

Let $(S, s_0, \delta_d, \delta_p)$ be a PTS as defined in Sec. 2.2. We say a state s' is *reachable* from s if there is a path $s \rightarrow_d^* s'$ under the deterministic transition relation. We say s is valid if s is reachable from s_0. Clearly if s is valid and s' is reachable from s then s' is itself valid. Throughout this work we only work with traces that only contain valid states. We use S_{valid} to denote the valid subset of S.

Let $\mathcal{R}$ be a binary relation over S. In Sec. 2.3 we defined two propositions $\Diamond_n(\mathcal{R})$ and $\Diamond_{n,p}(\mathcal{R})$. Recall that their definitions are as follows:

- $\Diamond_n(\mathcal{R})$: If e is any state element in a segmented trace, and e' is a state element that branched out from e in n time-steps, then $\mathcal{R}(e.\mathrm{s}, e'.\mathrm{s})$ holds.
- $\Diamond_{n,p}(\mathcal{R})$: Let e be any state element in a segmented trace. Let D be the state distribution that branched out from e in n time-steps. If e' is any state element in D, then $\mathcal{R}(e.\mathrm{s}, e'.\mathrm{s})$ holds with probability at least p.

It is clear that $\Diamond_n(\mathcal{R})$ implies $\Diamond_{n,1}(\mathcal{R})$. The converse is not true due to a technical issue: we allow state elements with probability 0. If we modify the definition of $\Diamond_n$ to exclude state elements with probability 0, then $\Diamond_n(\mathcal{R})$ would be equivalent to $\Diamond_{n,1}(\mathcal{R})$.

We shall use $\mathrm{Res}(\mathcal{R})$ to denote the restriction of $\mathcal{R}$ to pairs (s, s') with the following property: s is valid and s' is reachable from s. We shall use $\mathcal{R}_1 \bowtie \mathcal{R}_2$ to denote the *inner join* of $\mathcal{R}_1, \mathcal{R}_2$: a pair (s, s'') is in $\mathcal{R}_1 \bowtie \mathcal{R}_2$ iff there exists s' such that $(s, s') \in \mathcal{R}_1, (s', s'') \in \mathcal{R}_2$. The following lemmas show how to chain temporal guarantees.

Lemma 9. *If* $\mathrm{Res}(\mathcal{R}_1) \bowtie \mathrm{Res}(\mathcal{R}_2) \subseteq \mathcal{R}_3$, $\Diamond_n(\mathcal{R}_1)$, *and* $\Diamond_m(\mathcal{R}_2)$, *then* $\Diamond_{n+m}(\mathcal{R}_3)$.

Lemma 10. *If* $\mathrm{Res}(\mathcal{R}_1) \bowtie \mathrm{Res}(\mathcal{R}_2) \subseteq \mathcal{R}_3$, $\Diamond_{n,p}(\mathcal{R}_1)$, *and* $\Diamond_m(\mathcal{R}_2)$, *then* $\Diamond_{n+m,p}(\mathcal{R}_3)$.

Lemma 11. *If* $\mathrm{Res}(\mathcal{R}_1) \bowtie \mathrm{Res}(\mathcal{R}_2) \subseteq \mathcal{R}_3$, $\Diamond_n(\mathcal{R}_1)$, *and* $\Diamond_{m,p}(\mathcal{R}_2)$, *then* $\Diamond_{n+m,p}(\mathcal{R}_3)$.

Lemma 12. *If* $\mathrm{Res}(\mathcal{R}_1) \bowtie \mathrm{Res}(\mathcal{R}_2) \subseteq \mathcal{R}_3$, $\Diamond_{n,p}(\mathcal{R}_1)$, *and* $\Diamond_{m,q}(\mathcal{R}_2)$, *then* $\Diamond_{n+m,pq}(\mathcal{R}_3)$.

Recall that the "always" operator $\Box(P)$ can be defined as $\Diamond_0(s = s' \wedge P(s))$. The above lemmas can also be used to combine eventual and always guarantees. Let $\mathcal{T}$ denote the always-true relation or predicate. We also have two "axioms."

Lemma 13. $\Diamond_n(\mathcal{T})$ *and* $\Diamond_{n,p}(\mathcal{T})$ *hold for any* n *and any* p.

Lemma 14. $\Box(\mathcal{T})$ *always holds.*

We now state some lemmas on how logical operators interact with temporal operators.

Lemma 15. *If* $\Diamond_n(\mathcal{R}_1)$ *and* $\Diamond_n(\mathcal{R}_2)$ *both hold, then* $\Diamond_n(\mathcal{R}_1 \cap \mathcal{R}_2)$ *holds. More generally, if* $\{\mathcal{R}_i\}$ *is a set of relations indexed by* I, *and* $\Diamond_n(\mathcal{R}_i)$ *holds for every* $i \in I$, *then* $\Diamond_n(\bigcap_{i \in I} \mathcal{R}_i)$ *holds.*

Lemma 16. *If* $\Diamond_{n,p}(\mathcal{R}_1)$ *and* $\Diamond_{n,q}(\mathcal{R}_2)$ *both hold, then* $\Diamond_{n,\max(0,p+q-1)}(\mathcal{R}_1 \cap \mathcal{R}_2)$ *holds.*

Lemma 17. *If* $\Diamond_{n,p}(\mathcal{R}_1)$ *and* $\Diamond_{n,q}(\mathcal{R}_2)$ *both hold, then* $\Diamond_{n,\max(p,q)}(\mathcal{R}_1 \cup \mathcal{R}_2)$ *holds. More generally, if* $\{\mathcal{R}_i\}$ *is a set of relations indexed by* I, $f : I \mapsto \mathbb{Q}$ *is a function, and* $\Diamond_{n,f(i)}(\mathcal{R}_i)$ *holds for every* $i \in I$, *then* $\Diamond_{n,\max_{i \in I}\{f(i)\}}(\bigcup_{i \in I} \mathcal{R}_i)$ *holds.*

A.2 Expected Termination Time

Let $\mathcal{R}$ be a binary relation over S. Let $t(n) : \mathbb{N} \mapsto \mathbb{N}, f(n) : \mathbb{N} \mapsto \mathbb{Q}$ be two functions. Assume that $\Diamond_{t(n),f(n)}(\mathcal{R})$ holds for every $n \in \mathbb{N}$. Without loss of generality, we shall assume both $t(n)$ and $f(n)$ are monotonic. Let e be a state element in an infinite segmented trace. Let T represent the random

$$\frac{p_i \text{ honest} \quad u[r] = s \quad q \in \Gamma'}{u \longrightarrow_p [(u[r \leftarrow \delta_O(s,i,q)], 1)]} \text{ Opponent}$$

$$\frac{p_i \text{ honest} \quad u[r] = s \quad (s,(s',i,v)) \in \delta_P}{u \longrightarrow_p [(u[r \leftarrow s'], 1)]} \text{ Proponent}$$

$$\frac{u[r] = s \quad (s, D) \in \delta_L}{u \longrightarrow_p \mathsf{map}\ (e \mapsto (u[r \leftarrow e.\mathrm{s}], e.\mathrm{p}))\ D} \text{ Local}$$

Fig. 8. Semantics of Countably-Infinite Parallel Composition.

variable that indicates the number of time-steps passed, starting from e, before $\mathcal{R}(e.\mathrm{s}, e'.\mathrm{s})$ becomes true for the first time. Then recall the tail sum formula:

$$\mathbb{E}[T] = \sum_{i=1}^{\infty} P[T \geq i].$$

For each $i > t(k)$, clearly we have $P[T \geq i] \leq 1 - f(k)$. Therefore we have

$$\mathbb{E}[T] \leq t(0) + \sum_{k=0}^{\infty} [t(k+1) - t(k)] \cdot [1 - f(k)].$$

For example, the probabilistic termination result we proved for Alg. 7 has $t(n) = 7n + 21$ and $p(n) = 1 - 1/2^n \cdot (n+1)$. Then we have

$$\mathbb{E}[T] \leq 21 + \sum_{k=0}^{\infty} 7(k+1) \cdot 1/2^k = 49.$$

The constants we proved are not optimal and can be further improved, but they are sufficient to demonstrate that the protocol terminates in $O(1)$ expected time.

B More Details of Parallel and Layered Composition of Functionalities

B.1 Parallel Composition

Let $\mathcal{F}$ be a functionality with state space S, initial state s_0, signature (Γ, Θ), opponent moves δ_O, proponent moves δ_P, and local moves δ_L. We define its countably-infinite parallel composition $\bigotimes_{r=0}^{\infty} \mathcal{F}$ as follows. The state space is $\mathsf{FinMap}(\mathbb{N} \mapsto S)$. We shall use u to denote such a finite map. In the initial state u_0 we have $u_0[r] = s_0$ for every $r \in \mathbb{N}$. The signature is $(\mathbb{N} \times \Gamma, \mathbb{N} \times \Theta)$. The semantics of the composed functionality is shown in Fig. 8.

B.2 Proof of Lemma 2

If $\mathcal{R}$ is the refinement relation between $\mathcal{F}, \mathcal{G}$, then the refinement relation between $\bigotimes_{r=0}^{\infty} \mathcal{F}$ and $\bigotimes_{r=0}^{\infty} \mathcal{G}$ is that $\mathcal{R}$ holds between every pair of instance states $u[r]$ and $u'[r]$, where u, u' are states of $\bigotimes_{r=0}^{\infty} \mathcal{F}$ and $\bigotimes_{r=0}^{\infty} \mathcal{G}$ respectively.

Initially, each $u[r]$ and $u'[r]$ are the initial states of $\mathcal{F}, \mathcal{G}$, and $\mathcal{R}$ holds between them by assumption. At each step, only one of the instances is affected, while other instances are unchanged. Clearly the refinement relation still holds for the unaffected instances. For the affected instance, the refinement proof between $\mathcal{F}, \mathcal{G}$ translates the step at $\mathcal{G}$ to zero or more steps at $\mathcal{F}$.

B.3 Layered Composition

We now illustrate the semantics of layered composition. Suppose we have three functionalities $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$. The first overlay application $\mathcal{G}^{(1)}$ implements $\mathcal{F}_2$ using $\mathcal{F}_1$ as its underlay, and the second

overlay application $\mathcal{G}^{(2)}$ implements $\mathcal{F}_3$ using $\mathcal{F}_2$ as its underlay. Layered composition replaces the abstract underlay $\mathcal{F}_2$ seen by $\mathcal{G}^{(2)}$ with the concrete implementation $\mathcal{G}^{(1)}$. The resulting application, written as $\mathcal{G}^{(1)} \odot \mathcal{G}^{(2)}$, implements $\mathcal{F}_3$ directly on top of $\mathcal{F}_1$.

For $j \in \{1, 2, 3\}$, let $\mathcal{F}_j$ have interface (Γ_j, Θ_j), state space S_j, initial state $s_{j,0}$, and transition components $\delta_O^{\mathcal{F}_j}$, $\delta_P^{\mathcal{F}_j}$, and $\delta_L^{\mathcal{F}_j}$. For $k \in \{1, 2\}$, let $\mathcal{G}^{(k)}$ have local state space $S_{local}^{(k)}$, message space $\mathcal{M}^{(k)}$, and overlay transition components

$$\delta_O^{(k)}, \delta_P^{(k)}, \delta_{L,send}^{(k)}, \delta_{L,call}^{(k)}, \delta_{L,recv}^{(k)}, \delta_{L,return}^{(k)}, \delta_{L,prob}^{(k)}.$$

The composed system keeps the state of both overlays $\mathcal{G}^{(1)}, \mathcal{G}^{(2)}$ and the underlay $\mathcal{F}_1$:

$$x = (\ell_2, \mu_2, \ell_1, \mu_1, u).$$

Here $\ell_k : ID \mapsto S_{local}^{(k)}$ is the local-state map of the honest processes in $\mathcal{G}^{(k)}$, μ_k is the message state for $\mathcal{G}^{(k)}$, and $u \in S_1$ is the state of $\mathcal{F}_1$. Thus the state space is

$$\texttt{FinMap}(ID \mapsto S_{local}^{(2)}) \times S_{msg}^{(2)} \times \texttt{FinMap}(ID \mapsto S_{local}^{(1)}) \times S_{msg}^{(1)} \times S_1.$$

The initial state is $(\ell_{2,0}, \mu_{2,0}, \ell_{1,0}, \mu_{1,0}, s_{1,0})$. The external interface of the composed system is (Γ_3, Θ_3), namely the interface of the upper application $\mathcal{G}^{(2)}$. Its underlay interface is (Γ_1, Θ_1), namely the interface of the lower functionality $\mathcal{F}_1$.

Upper-layer steps. External calls and returns are handled exactly as in $\mathcal{G}^{(2)}$. They only affect the local state of the upper layer:

$$\frac{p_i \text{ honest} \quad \ell_2[i] = s \quad q \in \Gamma_3}{x \rightarrow_p [((\ell_2[i \leftarrow \delta_O^{(2)}(i, s, q)], \mu_2, \ell_1, \mu_1, u), 1)]}$$

$$\frac{p_i \text{ honest} \quad \ell_2[i] = s \quad ((i, s), (s', v)) \in \delta_P^{(2)}}{x \rightarrow_p [((\ell_2[i \leftarrow s'], \mu_2, \ell_1, \mu_1, u), 1)]}.$$

The ordinary message and local steps of $\mathcal{G}^{(2)}$ are also lifted componentwise:

$$\frac{p_i \text{ honest} \quad \ell_2[i] = s \quad ((i, s), (s', m)) \in \delta_{L,send}^{(2)}}{x \rightarrow_p [((\ell_2[i \leftarrow s'], \mu_2[\mathcal{M}_{sent} \mathrel{+}= m], \ell_1, \mu_1, u), 1)]}$$

$$\frac{p_i \text{ honest} \quad \ell_2[i] = s \quad m \in \mu_2.\mathcal{M}_{sent}}{x \rightarrow_p [((\ell_2[i \leftarrow \delta_{L,recv}^{(2)}(i, s, m)], \mu_2[\mathcal{M}_{deliv} \mathrel{+}= (i, m)], \ell_1, \mu_1, u), 1)]}$$

$$\frac{p_i \text{ honest} \quad \ell_2[i] = s \quad ((i, s), D) \in \delta_{L,prob}^{(2)}}{x \rightarrow_p \texttt{map}\,(e \mapsto ((\ell_2[i \leftarrow e.\mathrm{s}], \mu_2, \ell_1, \mu_1, u), e.\mathrm{p}))\, D}.$$

Connecting the two layers. The interesting part is what happens when $\mathcal{G}^{(2)}$ wants to call its underlay $\mathcal{F}_2$. In the composed system there is no separate abstract $\mathcal{F}_2$ state. Instead, the call is delivered to $\mathcal{G}^{(1)}$, which is precisely the implementation of $\mathcal{F}_2$. Thus an upper-layer call becomes an opponent step of the middle layer:

$$\frac{p_i \text{ honest} \quad \ell_2[i] = s \quad \ell_1[i] = t \quad ((i, s), (s', q)) \in \delta_{L,call}^{(2)}}{x \rightarrow_p [((\ell_2[i \leftarrow s'], \mu_2, \ell_1[i \leftarrow \delta_O^{(1)}(i, t, q)], \mu_1, u), 1)]}.$$

Conversely, when $\mathcal{G}^{(1)}$ returns a value through the interface of $\mathcal{F}_2$, that return is consumed by the upper layer's return handler:

$$\frac{p_i \text{ honest} \quad \ell_2[i] = s \quad \ell_1[i] = t \quad ((i,t),(t',v)) \in \delta_P^{(1)}}{x \rightarrow_p [((\ell_2[i \leftarrow \delta^{(2)}_{L,return}(i,s,v)], \mu_2, \ell_1[i \leftarrow t'], \mu_1, u), 1)]}.$$

These two rules are the whole purpose of layered composition: calls going down are redirected from the abstract middle functionality to its implementation, and returns coming back up are redirected from the implementation to the upper application.

Middle-layer steps. All other steps of $\mathcal{G}^{(1)}$ are ordinary overlay steps on top of $\mathcal{F}_1$. Sending and receiving middle-layer messages only update ℓ_1 and μ_1:

$$\frac{p_i \text{ honest} \quad \ell_1[i] = t \quad ((i,t),(t',m)) \in \delta^{(1)}_{L,send}}{x \rightarrow_p [((\ell_2, \mu_2, \ell_1[i \leftarrow t'], \mu_1[\mathcal{M}_{sent} \mathrel{+}= m], u), 1)]}$$

$$\frac{p_i \text{ honest} \quad \ell_1[i] = t \quad m \in \mu_1.\mathcal{M}_{sent}}{x \rightarrow_p [((\ell_2, \mu_2, \ell_1[i \leftarrow \delta^{(1)}_{L,recv}(i,t,m)], \mu_1[\mathcal{M}_{deliv} \mathrel{+}= (i,m)], u), 1)]}.$$

Middle-layer probabilistic local steps are lifted in the same way:

$$\frac{p_i \text{ honest} \quad \ell_1[i] = t \quad ((i,t),D) \in \delta^{(1)}_{L,prob}}{x \rightarrow_p \mathsf{map}\,(e \mapsto ((\ell_2, \mu_2, \ell_1[i \leftarrow e.\mathrm{s}], \mu_1, u), e.\mathrm{p}))\, D}.$$

When $\mathcal{G}^{(1)}$ calls its own underlay, the call is sent to the real lower functionality $\mathcal{F}_1$:

$$\frac{p_i \text{ honest} \quad \ell_1[i] = t \quad ((i,t),(t',q)) \in \delta^{(1)}_{L,call}}{x \rightarrow_p [((\ell_2, \mu_2, \ell_1[i \leftarrow t'], \mu_1, \delta_O^{\mathcal{F}_1}(u,i,q)), 1)]}.$$

If $\mathcal{F}_1$ returns a value, it is handled by $\mathcal{G}^{(1)}$:

$$\frac{p_i \text{ honest} \quad \ell_1[i] = t \quad (u,(u',i,v)) \in \delta_P^{\mathcal{F}_1}}{x \rightarrow_p [((\ell_2, \mu_2, \ell_1[i \leftarrow \delta^{(1)}_{L,return}(i,t,v)], \mu_1, u'), 1)]}.$$

Finally, local steps of $\mathcal{F}_1$ are lifted by updating only the lower underlay state:

$$\frac{(u,D) \in \delta_L^{\mathcal{F}_1}}{x \rightarrow_p \mathsf{map}\,(e \mapsto ((\ell_2, \mu_2, \ell_1, \mu_1, e.\mathrm{s}), e.\mathrm{p}))\, D}.$$

Byzantine message injections are included for either message space, exactly as in the overlay semantics of Fig. 6: injecting $m \in \mathcal{M}^{(k)}$ appends m to the sent-message set of μ_k and leaves every other component unchanged. The probabilistic transition relation of $\mathcal{G}^{(1)} \odot \mathcal{G}^{(2)}$ is the union of the rules above. No other transitions are allowed.

B.4 Proof of Lemma 3

Refinement between $\mathcal{G}^{(1)} \odot \mathcal{G}^{(2)}$ is proved via an intermediate refinement layer. We first show that $\mathcal{G}^{(1)} \odot \mathcal{G}^{(2)}$ refines $\mathcal{G}^{(2)}$. Let $\mathcal{R}_1$ be the refinement relation between $\mathcal{G}^{(1)}$ and $\mathcal{F}_2$. Then the refinement relation between $\mathcal{G}^{(1)} \odot \mathcal{G}^{(2)}$ and $\mathcal{G}^{(2)}$ is: the process-local states and the sent/delivered messages of $\mathcal{G}^{(2)}$ are identical, while $\mathcal{R}_1$ holds between the state of $\mathcal{G}^{(1)}$ and the state of $\mathcal{F}_2$.

It is already assumed that $\mathcal{G}^{(2)}$ refines $\mathcal{F}_3$, with refinement relation $\mathcal{R}_2$. Therefore the final refinement relation is: there exists a state of $\mathcal{G}^{(2)}$, such that refinement holds both between $\mathcal{G}^{(1)} \odot \mathcal{G}^{(2)}$ and $\mathcal{G}^{(2)}$, and between $\mathcal{G}^{(2)}$ and $\mathcal{F}_3$.

To see that $\mathcal{G}^{(1)} \odot \mathcal{G}^{(2)}$ refines $\mathcal{G}^{(2)}$: each upper-layer step affects only the state of $\mathcal{G}^{(2)}$, so the refinement between $\mathcal{G}^{(1)}$ and $\mathcal{F}_2$ still holds; each middle-layer step affects only the state of $\mathcal{G}^{(1)}$, so the state of $\mathcal{G}^{(2)}$ remains identical. Each call/return step between $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(2)}$ is refined by a call/return step between $\mathcal{F}_2$ and $\mathcal{G}^{(2)}$.

Received 2025-11-14; accepted 2026-04-03