Tight Conditions for Binary-Output Tasks Under Crashes

Timothé Albouy
IMDEA Software Institute, Madrid, Spain

Antonio Fernández Anta
IMDEA Software Institute, Madrid, Spain
IMDEA Networks Institute, Madrid, Spain

Chryssis Georgiou
University of Cyprus, Nicosia, Cyprus

Nicolas Nicolaou
Algolysis Ltd, Nicosia, Cyprus

Junlang Wang
IMDEA Networks Institute, Madrid, Spain
Universidad Carlos III de Madrid, Spain

Abstract

This paper explores necessary and sufficient system conditions to solve distributed tasks with binary outputs (i.e., tasks with output values in $\{0, 1\}$). We focus on the distinct output sets of values a task can produce (intentionally disregarding validity and value multiplicity), considering that some processes may output no value. In a distributed system with n processes, of which up to $t \leq n$ can crash, we provide a complete characterization of the tight conditions on n and t under which every class of tasks with binary outputs is solvable, for both synchronous and asynchronous systems. This output-set approach yields highly general results: it unifies multiple distributed computing problems, such as binary consensus and symmetry breaking, and it produces impossibility proofs that hold for stronger task formulations, including those that consider validity, account for value multiplicity, or move beyond binary outputs.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed solvability, Asynchrony, Synchrony, Impossibility proofs, Binary-output tasks, Crash tolerance, Disagreement

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2025.5

Funding This work has been partially supported by the Spanish Ministry of Science and Innovation under grants SocialProbing (TED2021-131264B-I00) and DRONAC (PID2022-140560OB-I00), the ERDF "A way of making Europe", NextGenerationEU, and the Spanish Government's "Plan de Recuperación, Transformación y Resiliencia". This work is part of the grant CEX2024-001471-M funded by MICIU/AEI/10.13039/501100011033.

1 Introduction

Distributed computing examines a wide range of coordination problems involving multiple processes interacting through a communication medium, such as a message-passing network or a shared memory. Many of these problems can be abstracted as distributed tasks, which can be seen as black boxes taking an input vector and producing an output vector. In these vectors, each process has (at most) one private input and one private output value. For example, the well-known *consensus* problem can be represented as a task with one input value per process (the proposals) and one output value per process (the decisions). In addition to this input/output interface, a task definition can also impose requirements on the following aspects:

- *Input constraint*: a constraint on the set of possible input vectors (which is especially relevant for *colored* tasks [15] and asymmetric problems, e.g., reliable broadcast [6]);
- *Output constraint*: a constraint on the set of possible output vectors (e.g., in consensus, all processes decide the same value);
- *Validity*: connection between the inputs and outputs (e.g., in consensus, the decided value must have been previously proposed by some process).

A central challenge in this field is to characterize the necessary and sufficient conditions under which distributed tasks can be solved, with many possible variations based on different assumptions on timing (asynchrony, synchrony) or failures (crashes, Byzantine faults, omissions). The celebrated FLP theorem exemplifies this approach by demonstrating that consensus is impossible in any asynchronous system with even one process crash [12].

Towards a unifying framework for task solvability. When exploring the boundary between solvable and unsolvable tasks in distributed computing, a strategic approach is to analyze the weakest, most general possible version of a task for which an impossibility result still holds. An impossibility proof for a weak task is particularly powerful because it automatically applies to all stronger versions of that same task.

Much of the literature on distributed computability has focused on particular tasks, such as consensus [12], renaming [8], set agreement [9], or election [19]. These studies have established powerful solvability and impossibility results, but they often rely on model-specific arguments (e.g., on the communication medium) or on constraints such as validity, which tie output to input values. This fragmentation makes it difficult to see the common computational structure underlying different tasks. A natural question, therefore, is whether there exists a unifying perspective that abstracts away from inputs and instead focuses on the essential combinatorial structure of task outputs.

Our approach. In this paper, we address this question for the case of *binary-output* tasks (i.e., whose output values belong to $\{0, 1\}$) in *crash-prone* systems. We introduce a classification based only on the *sets of distinct output values* that executions of a binary-output task T may produce, disregarding multiplicity in the output values. More formally, we say that a single execution of T produces an *output set* included in $\{0, 1\}$, considering that some processes may output no value (possibly intentionally). There are only four possible binary output sets produced by an execution of T: $\varnothing$, $\{0\}$, $\{1\}$, and $\{0, 1\}$. Hence, the *set of output sets* of T is defined as the set $O \subseteq \{\varnothing, \{0\}, \{1\}, \{0, 1\}\}$ of all possible output sets that can be produced across all the executions of T (by providing all possible inputs).

This very weak abstraction allows us to strip tasks down to their minimally observable outcomes and ask: given the system size n and crash tolerance t, which sets of output sets are achievable? Or, conversely, given a set of output sets O, which combinations of n and t make O achievable? Our main result is a complete characterization of the tight conditions on n and t that determine solvability for all possible sets of output sets O, both in synchronous and asynchronous models.

The fact that our framework abstracts away the inputs and only focuses on the output sets of binary-output tasks, implies that impossibility proofs established are directly applicable to a broader range of stronger, more constrained tasks, including validity-based and multi-valued tasks. This shift in focus does not just simplify the analysis; it also allows for the generalization of fundamental necessary conditions that apply to stronger task formulations. Moreover, our framework remains expressive enough to capture a wide array of tasks, from classic ones like binary consensus to novel ones such as a new form of symmetry breaking.

It is important to note that, while our approach excels at generalizing impossibility results, the specific system conditions on n and t we identify in this paper are guaranteed to be tight only for binary-output tasks without validity restrictions. Any additional constraint, such as a validity condition or a requirement on value multiplicity, may necessitate a stronger condition on the system parameters to achieve solvability.

Contributions and roadmap. Our main contributions are the following.

1. *A novel framework for studying binary-output tasks*: We introduce a new methodology striving to unify all distributed tasks with binary output values. More precisely, we focus on the sets of distinct output bits that can be produced by these tasks in crash-prone environments, abstracting away the input values, the output multiplicity, or the communication medium (message passing or shared memory). As a result, the formalization that we obtain (see Section 3) is quite simple (it only relies on combinatorial techniques), thus facilitating formal reasoning, while staying expressive enough to capture many significant families of distributed tasks.
2. *A complete solvability characterization*: We exhaustively examine all 16 possible combinations of distinct output bits that binary-output tasks can produce, and we provide the tight conditions to implement each of them (see Section 4, Table 2), both in the asynchronous and synchronous cases, by proving both their necessity (see Section 5), via impossibility proofs, and their sufficiency (see Section 6), via algorithms and correctness proofs.
3. *New symmetry-breaking problems*: As an interesting twist, our novel way to classify binary-output tasks allowed us to discover new interesting problems, in particular, one that we baptized *disagreement* (different from classical weak/strong symmetry breaking), which must always guarantee that the system does not agree on one single output value (see Section 6).

Section 2 exposes the research landscape in which this work is situated, and concluding remarks are provided in Section 7. For the sake of presentation clarity, Appendix A contains omitted algorithms and proofs.

2 Related Work

Our work strives to classify and characterize many classes of distributed tasks. In this section, we first present the tasks that are relevant to our characterization (along with some of their most salient (un)solvability results), and we then review the existing endeavors attempting to unify entire families of tasks under the same framework.

The two most studied (non-distinct) families of distributed tasks are *agreement* and *symmetry breaking*. Informally, in agreement, we want to constrain the maximum number of different outputs, while in symmetry breaking we want to do the opposite, i.e., constrain the minimum number of different outputs.

Agreement. The agreement family includes tasks such as k-set agreement [9], reliable broadcast [6], or consensus [20]. In particular, consensus is a fundamental agreement abstraction of distributed computing, where all participants propose a value and must eventually agree on one of the proposed values. A foundational impossibility result for this problem is the FLP theorem [12], which shows that consensus is impossible in an asynchronous system with even one process crash. In this context, Mostéfaoui, Rajsbaum, and Raynal explored how conditions on the input vector can make consensus solvable in asynchronous

crash-prone systems [21]. In a way, we take the opposite approach: instead of characterizing the solvability of a task based on its *input*, we focus on its *output* (yielding necessary conditions that also hold if we account for inputs). Aside from the asynchronous case, some other works also study the solvability of agreement tasks in synchronous systems [22, 24].

Symmetry breaking. The symmetry-breaking family includes tasks such as election [17, 25], renaming [8], and weak/strong symmetry breaking (WSB/SSB) [17]. In an election, only one process outputs 1, while all others output 0. In k-renaming, k processes in the system output distinct values (corresponding to their new identities). Notice that our binary-output approach maps to 2-renaming, but not >2-renaming. WSB guarantees that 0 and 1 are output only if all processes are correct, and SSB adds the property that 1 is always output (even if there are faults). Some solvability results also exist for symmetry breaking, e.g., on the wait-free solvability of WSB and renaming [23].

Unification attempts. Several endeavors before us have attempted to unify multiple classes of distributed tasks under the same framework, in order to draw general results from their common structure. For instance, *generalized symmetry breaking* (*GSB*) aims to unify many symmetry-breaking tasks [7, 17], such as election, renaming, and weak symmetry-breaking. Furthermore, many works employ a topological approach to study the computability of distributed tasks. With topology, tasks are modeled as input and output complexes, and solvability is characterized by the existence of continuous maps, with impossibility linked to invariants such as connectivity. One of the most prominent results in this area is the *asynchronous computability theorem* (*ACT*), which characterizes the solvability of *wait-free* tasks in an asynchronous shared-memory model [16]. The *musical benches* problem also leverages topological techniques to capture 2-set agreement and 2-renaming under the same abstraction in a wait-free shared-memory context [13]. For more material on the applications of combinatorial topology to distributed computing, we refer the interested reader to Herlihy, Kozlov, and Rajsbaum's monograph [15].

3 Computing Model and Problem Formalization

For clarity, we provide in Table 1 a list of concepts and notations used in this paper.

3.1 Computing Model

Process model. We consider a distributed system with a set $P = \{p_1, p_2, ..., p_n\}$ of processes ($|P| = n$). Processes are deterministic computing entities that take steps according to their local state and the events they observe, following an algorithm A. Failures are restricted to *crash faults*: in an execution of the system a process may halt prematurely and take no further steps, but it does not deviate from its algorithm before crashing. We assume an upper bound t on the number of processes that may crash in an execution, with $0 \leq t \leq n$. A process that does not crash in an execution E is said to be *correct* in E.

Communication model. Processes interact through a generic communication medium. This medium is *reliable* as far as it does not suppress, duplicate, or corrupt information. Every process $p \in P$ has access to two abstract operations that capture the behavior of this communication medium:

- communicate I: process p disseminates some information I to the system processes;
- observe I (callback event): process p is notified that information I was communicated.

Table 1 Concepts and notations used in this paper.

Concept or notation	Meaning
p_i	process of the system with identity i
P	Set of processes in the system
n	Number of processes in the system ($0 \leq n = \|P\|$)
t	Upper bound on the number of crashed processes ($0 \leq t \leq n$)
f	Effective number of crashed processes in a run ($0 \leq f \leq t$)
$\sigma \in \{\text{Sync}, \text{Async}\}$	Timing model (synchronous or asynchronous)
PRNG	Pseudo-random number generation
pseudo_random_pick(S)	Function returning a pseudo-random value v from set S
T, A, E	Task, algorithm, execution
$V_{\text{in}} \in (\mathbb{I} \cup \{\perp\})^n$	Input vector
$V_{\text{out}} \in (\mathbb{O} \cup \{\perp\})^n$	Output vector, with $\mathbb{O} = \{0, 1\}$
$\perp$	Sentinel value denoting no input/output
$\star$	Unspecified value
no_out	Boolean indicating if $\varnothing$ is a possible output set or not

The length of the information I is not bounded. From a terminology point of view, we say that processes *communicate* and *observe* information. The medium guarantees that all correct processes eventually obtain a consistent view of the set of communicated information, while crashed processes may only have partial but always valid views. More formally, the communication abstraction satisfies the following properties (the "C" prefix stands for "communication").

- **C-Validity:** If a process p observes information I, then I must have been previously communicated by some process p'.
- **C-Local-Termination:** If a *correct* process p communicates information I, then some *correct* process p' (if there is any) eventually observes I.
- **C-Global-Termination:** If a process p observes information I, then all *correct* processes eventually observe I.

C-Validity is a safety property, while C-Local-Termination and C-Global-Termination are liveness properties. These communicate/observe operations can be implemented straightforwardly on top of classic communication media such as message-passing networks or shared memory, regardless of the number of crashes t or the timing assumptions (synchronous, asynchronous, etc.). For instance, in message passing, they can be realized using reliable broadcast [14] (implementable under synchrony and asynchrony with any $t \leq n$), which ensures consistency of observations. They can also be implemented from weak shared memory models, such as eventually consistent registers [26]: each process writes its communicated information into a new register, while other processes periodically scan all registers to detect new observable information.

Timing models: Async and Sync. As usual, we assume the existence of a global clock, to which processes have no access. We consider two classical timing models: asynchrony and synchrony, respectively denoted Async and Sync. For simplicity, we assume in both models that local computation occurs instantaneously, while only communication takes time.

In Async, the speed of information propagation is arbitrary but positive: if a correct process communicates some information I, then all correct processes eventually observe I, but the delay before the observation may be unbounded.

In Sync, communication and process execution proceed in globally coordinated lock-step rounds. Each round consists of two steps: first, a *communication step*, where processes communicate and observe information, and a *computation step*, where processes can perform actions such as outputting values [22]. Importantly, all information communicated during the communication step of a round is observed in that step, before the computation step of the same round begins. Therefore, in the Sync model, the communication medium provides the following additional liveness property.

- **C-Synchrony:** If an arbitrary process p communicates some information I at the start of synchronous round R, then all processes that observe I do so during the communication step of round R.

Pseudo-randomness. The algorithms presented in this paper rely on pseudo-random number generation (PRNG). Importantly, the use of PRNG does not strengthen the computing model: PRNG can be implemented deterministically given a local seed, which can, for instance, be derived from the local time of processes [18, p. 184]. The correctness of our algorithms requires that the generated numbers are different across all executions (a perfectly uniform PRNG distribution is not necessary). For that, it is enough that local seeds sometimes change from one execution to another. In the algorithms of this paper, we rely on the pseudo_random_pick(V) function that, given a set of values V, returns a value $v \in V$ pseudo-randomly.

3.2 Problem Formalization

Outputs. In an execution, a process $p \in P$ outputs a value $v \in \{0, 1\}$ using the operation output v. In a given execution, each process $p \in P$ outputs *at most one* value. Hence, all its invocations of the output operation must have the same value.

We often use the XOR operation to obtain the one's complement of a bit $v \in \{0, 1\}$. The logical formula we use is $1 \oplus v$, which gives 1 if $v = 0$, and 0 if $v = 1$.

Tasks. We define a task T as a relation between input vectors $V_{\text{in}} = (v^1_{\text{in}}, ..., v^n_{\text{in}})$ and output vectors $V_{\text{out}} = (v^1_{\text{out}}, ..., v^n_{\text{out}})$, where v^i_{in} and v^i_{out} are the input and output values of process p_i, respectively. That is, we have $(V_{\text{in}}, V_{\text{out}}) \in T$ if and only if V_{out} is a possible output of the task when the input is V_{in}. The elements v^i_{in} of an input vector V_{in} belong to the set $\mathbb{I} \cup \{\perp\}$, where $\mathbb{I}$ is the input alphabet and $\perp$ is a sentinel value (not in $\mathbb{I}$) that represents no input. Similarly, the elements v^i_{out} of an output vector V_{out} belong to the set $\mathbb{O} \cup \{\perp\}$, where $\mathbb{O}$ is the output alphabet and $\perp$ is a special value (not in $\mathbb{O}$) that represents no output. In this work we consider only *binary-output* tasks, i.e., $\mathbb{O} = \{0, 1\}$. Observe that a task T can have several possible outputs for the same input. By abuse of notation, we denote the set of all outputs V_{out} of T for an input V_{in} as $T(V_{\text{in}}) = \{V_{\text{out}} \mid (V_{\text{in}}, V_{\text{out}}) \in T\}$.

Output sets. In this work, we will not differentiate output vectors that contain the same set of different output values. For that, we use the following notation for the set of distinct values in an output vector V_{out}, which we call an *output set*: $OS(V_{\text{out}}) = \{v^i_{\text{out}} \in V_{\text{out}} \mid v^i_{\text{out}} \neq \perp\}$. Since we consider only binary-output tasks, it holds that $OS(V_{\text{out}}) \in 2^{\{0,1\}} = \{\varnothing, \{0\}, \{1\}, \{0, 1\}\}$.

We then define the *set of output sets* of a pair task/input vector (T, V_{in}) as the set of all possible output sets that can be produced by executing T with V_{in}: $SOSI(T, V_{\text{in}}) = \{OS(V_{\text{out}}) \mid V_{\text{out}} \in T(V_{\text{in}})\}$. Observe that $SOSI(T, V_{\text{in}}) \subseteq \{\varnothing, \{0\}, \{1\}, \{0, 1\}\}$, and that $SOSI(T, V_{\text{in}}) = \varnothing$ occurs when $T(V_{\text{in}}) = \varnothing$, because there is no V_{out} such that $(V_{\text{in}}, V_{\text{out}}) \in T$.

Finally, we define the *set of output sets* of a task T as the set of all possible output sets that T can produce with all possible input vectors V_{in}: $SOS(T) = \{OS(V_{\text{out}}) \mid \exists V_{\text{in}} \in (\mathbb{I} \cup \{\perp\})^n : V_{\text{out}} \in T(V_{\text{in}})\}$. Observe again that $SOS(T) \subseteq \{\varnothing, \{0\}, \{1\}, \{0,1\}\}$. Hence, all tasks T can be classified into 16 classes, each characterized by which subset of $\{\varnothing, \{0\}, \{1\}, \{0,1\}\}$ the set of output sets $SOS(T)$ is.

Algorithms and executions. An *algorithm* A defines the steps taken by each process $p \in P$ as a function of its local state (including its input) and the communication observed. The algorithm A combined with input V_{in} is denoted $A(V_{\text{in}})$. An *execution* E of $A(V_{\text{in}})$ is a (finite or infinite) sequence of steps taken by processes following algorithm A with input V_{in}, combined with the observation of communication events. The *output* of an execution E of $A(V_{\text{in}})$ is the vector V_{out} with the output value that is produced in E by each process p, or $\perp$ if no value is output by the process.

Failure and communication patterns. A *failure pattern* defines in an execution the identity of faulty processes and the time instant each faulty process stops taking steps. The possible failure patterns in an execution depend on n and the value of $f \leq t$ when defined, which is the exact number of failures. We denote the set of all failure patterns for a given pair (n, f) by $FP(n, f)$.

A *communication-delay pattern* defines the delay experienced by each communication in the execution. The possible communication-delay patterns in an execution depend on the synchrony ($\sigma \in \{\text{Async}, \text{Sync}\}$) of the system. We denote the set of all communication-delay patterns for a given σ by $CDP(\sigma)$.

Since we only consider deterministic algorithms and local computation is instantaneous, an execution is fully characterized by the tuple $(A, V_{\text{in}}, sd, fp, cdp)$, where A is the algorithm, V_{in} the input vector, sd is the seed of the PRNG, fp is the failure pattern, and cdp is the communication-delay pattern.

Implementation of a set of output sets. A *system configuration* is a triple $C = (n, t, \sigma)$, where n is the number of processes, t is the maximum number of crashed processes in any execution E, and $\sigma \in \{\text{Async}, \text{Sync}\}$ defines the timing model of the distributed system in which E runs.

▶ **Definition 1** (Implementation of a set of output sets). *Algorithm A* implements *the non-empty set of output sets $O \subseteq \{\varnothing, \{0\}, \{1\}, \{0,1\}\}$ under system configuration $C = (n, t, \sigma)$ with $\sigma \in \{Async, Sync\}$ if for all $0 \leq f \leq t$, we have the following properties.*

- ***Safety:*** *For all V_{in}, all possible PRNG seeds sd, all failure patterns $fp \in FP(n, f)$, and all communication-delay patterns $cdp \in CDP(\sigma)$, all executions (A, V_{in}, sd, fp, cdp) of $A(V_{in})$ in a system with n processes, f failures, and timing model σ, have output vectors V_{out} such that $OS(V_{out}) \in O$.*
- ***Completeness:*** *For each $o \in O$, there is a V_{in}, a PRNG seed sd, a failure pattern $fp \in FP(n, f)$, and a communication-delay pattern $cdp \in CDP(\sigma)$, such that execution (A, V_{in}, s, fp, cdp) of $A(V_{in})$ has an output vector V_{out} where $OS(V_{out}) = o$.*

▶ **Definition 2** (Binary-output task solvability). *Given a binary-output task T with non-empty set of output sets $SOS(T)$, task T can be solved under system configuration $C = (n, t, \sigma)$ with $\sigma \in \{Async, Sync\}$ only if there is an algorithm A that implements $SOS(T)$ under that configuration.*

In the above definitions, we exclude the degenerate case where the set of output sets is empty.

Table 2 Characterization of the tight conditions for implementing the 16 sets of output sets.

#	Exact set of output sets O	Timing model	Tight solvability condition	Necessity proof	Sufficiency proof
1	$\varnothing, \{0\}, \{1\}, \{0,1\}$	Async & Sync	$n \geq t, n \geq 2$	Observation 3	Algorithm 3 / Theorem 16 with V={0, 1, ⊥}
2	~~$\varnothing$~~, $\{0\}, \{1\}, \{0,1\}$	Async & Sync	$n > t, n \geq 2$	Observation 3	Algorithm 3 / Theorem 16 with V={0, 1}
3	$\varnothing$, ~~$\{0\}$~~, $\{1\}, \{0,1\}$	Async & Sync	$n \geq t, n \geq 2$	Observation 3	Algorithm 5 / Theorem 18 with v=1, *no_out*=`true`
4	~~$\varnothing$~~, ~~$\{0\}$~~, $\{1\}, \{0,1\}$	Async & Sync	$n > t, n \geq 2$	Observation 3	Algorithm 5 / Theorem 18 with v=1, *no_out*=`false`
5	$\varnothing, \{0\},$ ~~$\{1\}$~~, $\{0,1\}$	Async & Sync	$n \geq t, n \geq 2$	Observation 3	Algorithm 5 / Theorem 18 with v=0, *no_out*=`true`
6	~~$\varnothing$~~, $\{0\},$ ~~$\{1\}$~~, $\{0,1\}$	Async & Sync	$n > t, n \geq 2$	Observation 3	Algorithm 5 / Theorem 18 with v=0, *no_out*=`false`
7	$\varnothing$, ~~$\{0\}$~~, ~~$\{1\}$~~, $\{0,1\}$	Async	$n > \frac{3}{2}t + 1, n \geq 2$	Observation 3 & Theorem 5	Algorithm 1 / Theorem 10 with *no_out*=`true`
		Sync	$n \geq t + 2, n \geq 2$	Observation 3 & Theorem 4	Algorithm 2 / Theorem 15 with *no_out*=`true`
8	~~$\varnothing$~~, ~~$\{0\}$~~, ~~$\{1\}$~~, $\{0,1\}$	Async	$n > \frac{3}{2}t + 1, n \geq 2$	Observation 3 & Theorem 5	Algorithm 1 / Theorem 10 with *no_out*=`false`
		Sync	$n \geq t + 2, n \geq 2$	Observation 3	Algorithm 2 / Theorem 15 with *no_out*=`false`
9	$\varnothing, \{0\}, \{1\},$ ~~$\{0,1\}$~~	Async & Sync	$n \geq t, n \geq 1$	Observation 3	Algorithm 4 / Theorem 17 with *no_out* = `true`
10	~~$\varnothing$~~, $\{0\}, \{1\},$ ~~$\{0,1\}$~~	Async	$t = 0, n \geq 1$	Observation 3 & [2, 12]	Algorithm 4 / Theorem 17 with *no_out*=`false`
		Sync	$n > t, n \geq 1$	Observation 3	Algorithm 6 / Theorem 19
11	$\varnothing$, ~~$\{0\}$~~, $\{1\},$ ~~$\{0,1\}$~~	Async & Sync	$n \geq t, n \geq 1$	Observation 3	Algorithm 3 / Theorem 16 with V={1, ⊥}
12	~~$\varnothing$~~, ~~$\{0\}$~~, $\{1\},$ ~~$\{0,1\}$~~	Async & Sync	$n > t, n \geq 1$	Observation 3	Algorithm 3 / Theorem 16 with V={1}
13	$\varnothing, \{0\},$ ~~$\{1\}$~~, ~~$\{0,1\}$~~	Async & Sync	$n \geq t, n \geq 1$	Observation 3	Algorithm 3 / Theorem 16 with V={0, ⊥}
14	~~$\varnothing$~~, $\{0\},$ ~~$\{1\}$~~, ~~$\{0,1\}$~~	Async & Sync	$n > t, n \geq 1$	Observation 3	Algorithm 3 / Theorem 16 with V={0}
15	$\varnothing$, ~~$\{0\}$~~, ~~$\{1\}$~~, ~~$\{0,1\}$~~	Async & Sync	$n \geq t, n \geq 0$	Observation 3	Algorithm 3 / Theorem 16 with V={⊥}
16	~~$\varnothing$~~, ~~$\{0\}$~~, ~~$\{1\}$~~, ~~$\{0,1\}$~~	Async & Sync	N/A	N/A	N/A

4 Tight Solvability Conditions for Binary-Output Tasks

In this section, we introduce Table 2, which presents an exhaustive characterization of the tight conditions to implement all 16 possible binary sets of output sets, providing a necessary condition to solve their corresponding tasks. The first column is the line index. The second column corresponds to the set of output sets (the crossed sets in red are the forbidden output sets). The third column specifies the timing model assumed (Async, Sync, or any model if the tightness proof does not depend on timing assumptions). The fourth column provides the tight (i.e., necessary and sufficient) system condition on n (the system's size) and t (the crash-resilience) to implement the given set of output sets under the corresponding timing model. The fifth column refers to the necessity proof of the tight condition, while the sixth column refers to its sufficiency proof.

We can observe that many well-studied tasks map to a set of output sets in Table 2. Line 10 maps to the set of output sets of classical binary consensus (i.e., each execution outputs either 0 or 1) [12], while line 9 maps to abortable [5, 10] or randomized [4] binary consensus (i.e., some executions may never terminate and output a value). Line 8 maps to 2-renaming [8] (i.e., renaming of only 2 processes in the system). Line 2 maps to 2-set

agreement [9] (up to 2 different values can be output by the system). Line 4 (and its symmetric counterpart, line 6) maps to strong symmetry-breaking [3]: 1 is always output, but 0 can also be output in favorable conditions (e.g., all processes participate, or there are no crashes).

Remark that we prove the necessity of the condition $t = 0$ to solve the Async case of line 10 (implementing the set of output sets $\{\{0\},\{1\}\}$) by relying on the FLP theorem [12]. However, unlike the original FLP paper, which assumes a message-passing network, our model does not assume a particular communication medium. For this reason, we rely on the medium-agnostic impossibility proof of asynchronous resilient consensus presented in [2].

5 Necessity: Impossibility Proofs

In this section, we prove the necessity of the tightness conditions of Table 2, by showing the impossibility of implementing the corresponding sets of output sets when these conditions are not satisfied.

▶ **Observation 3.** *Under any timing model (Async or Sync), the following conditions are necessary to implement a set of output sets O:*

$$n \geq \max(\{|o| : o \in O\}), \quad \textit{and} \quad n - t \geq \min(\{|o| : o \in O\}).$$

Proof. Let us consider a set of output sets O. Condition $n \geq \max(\{|o| : o \in O\})$ is necessary for implementing O, otherwise there would not be enough processes to output all the values of the largest $o \in O$, violating completeness. Condition $n - t \geq \min(\{|o| : o \in O\})$ is also necessary for implementing O. Otherwise, when $f = t$, there would not be enough processes guaranteed to stay correct to output all the values of the smallest $o \in O$, violating safety. ◀

▶ **Theorem 4.** *Under any timing model (Async or Sync), the condition $n - t \geq 2$ is necessary for implementing the set of output sets $O = \{\varnothing, \{0,1\}\}$.*

Proof. We prove the lemma by contradiction: assume that there is an algorithm A that can implement $O = \{\varnothing, \{0,1\}\}$ under $n - t < 2$. This means that at least $n - 1$ processes can crash during an execution.

Let us consider an execution E_c of A that produces output set $\{0,1\}$. This execution must exist for completeness. In particular, let us consider the first process $p \in P$ that outputs some value $v \in \{0,1\}$ in E_c with respect to the global time, and let τ be the time when p outputs v.[1] Then, it is possible to craft an execution E that is exactly the same as E_c up to time τ, at which time all processes except p crash before outputting anything, and p is correct. In execution E, the output set is $\{v\}$, which violates safety: contradiction. ◀

▶ **Theorem 5.** *Under Async, condition $n > \frac{3}{2}t + 1$ is necessary for implementing the sets of output sets $O = \{\{0,1\}\}$ and $O' = \{\varnothing, \{0,1\}\}$.*

Proof. By contradiction, let us assume that there exists an algorithm A that implements the set of output sets $\{\{0,1\}\}$ or $\{\varnothing, \{0,1\}\}$ under Async and $n \leq \frac{3}{2}t + 1$. Note that $n \geq 2$, from Observation 3, which implies that $t \geq 1$.

We first prove that, no matter the case (whether A implements $\{\{0,1\}\}$ or $\{\varnothing, \{0,1\}\}$), A must have a crash-free execution E_0 producing the output set $\{0,1\}$.

[1] If multiple processes simultaneously output a value at time τ, we pick any of these processes, as their respective outputs cannot have physically influenced each other.

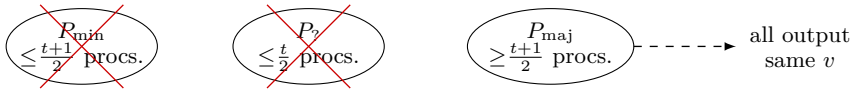

Figure 1 Execution E_{crash}, where processes of $P_{\min}$ and $P_?$ crash, and only processes of P_{maj} stay correct, which leads to a safety violation.

- If A implements the set of output sets $\{\{0,1\}\}$, then any crash-free execution of A can only output set $\{0,1\}$.
- If A implements the set of output sets $\{\varnothing,\{0,1\}\}$, then we consider any execution E_c of A that outputs set $\{0,1\}$ but that may present crashes. Let us denote by P_d the set of processes that output any value in E_c, and by P_c the set of processes that crash in E_c. We can construct an execution E_0 of A that is the same as E_c, except that it has no crash, and all information communicated by processes of P_c after the point where they would crash in E_c is delayed due to asynchrony, such that it is only observed by processes of P_d after they produce their outputs. Therefore, the outputs of the processes of P_d cannot have been influenced by the fact that the processes in P_c are correct, and E_0 must also produce the output set $\{0,1\}$.

We therefore refer to E_0 as a crash-free execution of A producing the output set $\{0,1\}$. Let $p_1 \in P$ be the first process that outputs some value $v_1 \in \{0,1\}$ in E_0 w.r.t. global time (at time τ_1). Let execution E_1 be an execution that is the same as E_0 up to τ_1, but where p_1 crashes immediately after it outputs v_1.

We now construct a sequence of executions $E_2, ..., E_t$ of A inductively as follows. Let us consider some $i = 2, ..., t$. To construct E_i, we assume by induction hypothesis that, in execution E_{i-1} (starting with E_1), process p_1 outputs value v_1, therefore E_{i-1} must necessarily produce the output set $\{0,1\}$ (it cannot produce the $\varnothing$ output set anymore). Hence, there must be another process in the system that outputs the opposite value $1 \oplus v_1$ in E_{i-1}. In particular, there must be some process $p_i \in P$ which is the second one (after p_1) to output a value $v_i \in \{0,1\}$ in E_{i-1} (w.r.t. global time) at time τ_i. We then make execution E_i the same as E_{i-1} until time τ_i, but we make process p_i crash just before it outputs a value, leaving p_1 be the only process that outputs a value in E_i up to time τ_i.

When we reach execution E_t, there have been t process crashes, so we cannot crash processes anymore. Finally, we denote by p_{t+1} the second process that outputs some value in E_t at time τ_{t+1}, w.r.t. global time.

We now create a new crash-free execution $E_{\text{crash-free}}$ that is the same as E_t, except that all processes $p_1, ..., p_t$ do not crash, but all communication that they make after their respective output is delayed until τ_{t+1}. Then, for any process $p \in P \setminus \{p_1, ..., p_t\}$, prior to τ_{t+1}, $E_{\text{crash-free}}$ is indistinguishable from E_t from the point of view of p. In other words, in $E_{\text{crash-free}}$, the processes of $\{p_1, ..., p_t\}$ are indistinguishable from crashed processes to the other processes, before τ_{t+1}.

Let us consider the set of processes $\{p_1, ..., p_{t+1}\}$ in $E_{\text{crash-free}}$. We can divide this set into two partitions: the subset P_0 of processes that output 0 in $E_{\text{crash-free}}$, and the subset P_1 of processes that output 1 in $E_{\text{crash-free}}$. Let us denote by $P_{\min}$ (resp., P_{maj}) the smaller (resp., bigger) set between P_0 and P_1 (if they have the same size, then $P_{\min}$ is P_0 and P_{maj} is P_1). Remark that $|P_{\min}| + |P_{\text{maj}}| = t + 1$ and $|P_{\min}| \leq \frac{t+1}{2} \leq |P_{\text{maj}}|$. We then denote by $P_?$ the subset of processes that are not in $P_{\min}$ or P_{maj}: $P_? = P \setminus \{p_1, ..., p_{t+1}\}$. We have $|P_?| = n - t - 1 \leq \frac{3}{2}t + 1 - t - 1 = \frac{t}{2}$. Moreover, we have $|P_{\min}| + |P_?| \leq \frac{t+1}{2} + \frac{t}{2} = t + \frac{1}{2}$. But as $|P_{\min}|$, $|P_?|$, and t are all integers, then we also have $|P_{\min}| + |P_?| \leq t$.

As illustrated in Figure 1, we can finally construct another execution E_{crash}, which is the same as $E_{\text{crash-free}}$ up to time τ_{t+1}, except that we make all processes of $P_{\min}$ and $P_?$ crash in E_{crash} immediately before their output step (as $|P_{\min}| + |P_?| \leq t$). After time τ_{t+1}, the only remaining correct processes in E_{crash} are those of P_{maj}. The processes of P_{maj} all output the same value v ($v = 0$ if $P_{\text{maj}} = P_0$, $v = 1$ if $P_{\text{maj}} = P_1$), thus producing the output set v. Hence, safety is violated, and we have a contradiction. ◀

6 Sufficiency: Implementing Algorithms

In this section, we provide algorithms, along with their correctness proofs, implementing the binary sets of output sets of Table 2. For modularity and conciseness, the algorithms of this paper feature *instantiation parameters*, which allow each one of them to have several possible behaviours, and thus implement multiple sets of output sets in Table 2.

For the sake of presentation clarity, some algorithms and proofs used in Table 2 are presented in detail in Appendix A. This section focuses on the algorithms implementing lines 7 and 8 of Table 2 (Algorithm 1 for the Async case, Algorithm 2 for the Sync case), which we deem the most interesting ones in our classification. We call them *disagreement* algorithms, since they must always guarantee that the system never outputs only one single value (0 or 1). More specifically, in every execution of a disagreement algorithm, there must be at least one process that outputs 0 and at least one that outputs 1.

6.1 Asynchronous Disagreement Algorithm (Lines 7–8)

The algorithm of this section (Algorithm 1) addresses the Async case of lines 7 and 8 in Table 2. It is instantiated by providing a Boolean $no\_out \in \{\texttt{true}, \texttt{false}\}$ (line 1), which determines if the empty output set $\varnothing$ can be produced or not.

At the initialization of the algorithm, the set of processes P is partitioned into three subsets, $P_0, P_1, P_?$ (line 2), which respectively contain at least $\frac{t+1}{2}$, $\frac{t}{2}$, and $\frac{t+1}{2}$ processes. By combining this with the system assumption, we obtain that the union of any two of these subsets contains at least one correct process. Moreover, a set of at least $t+1$ processes $P_{\text{init}} \subset P$ is also selected (which is only used when $no\_out = \texttt{true}$).

Every process $p_{\text{init}} \in P_{\text{init}}$ passes the condition at line 4 only if $\varnothing$ is not a forbidden output set ($no\_out = \texttt{true}$) and it pseudo-randomly picks 0. If it does, p_{init} communicate INIT. Every process $p_v \in P_v, v \in \{0,1\}$ first checks if $no\_out = \texttt{true}$ (line 6). If it does, p_v waits until it observes some INIT information. If p_v passes the previous line, it will then output v (line 7) and communicate OUTPUT(v) (line 8). Every process $p_? \in P_?$ first waits until it observes some OUTPUT(v) (line 10). If it passes this wait statement, then $p_?$ outputs the opposite of v, namely $1 \oplus v$ (line 11).

Before proving the correctness of Algorithm 1, we first show some intermediary results.

▶ **Observation 6.** *Since P_0 and P_1 are disjoint, we have $|P_0 \cup P_1| \geq \frac{t+1}{2} + \frac{t}{2} = t + \frac{1}{2}$. Moreover, as the left-hand side is an integer, then we also have $|P_0 \cup P_1| \geq t+1$, hence there is at least one correct process $p_v^c \in P_0 \cup P_1$.*

▶ **Lemma 7.** *If some process $p \in P_?$ outputs a value $v \in \{0,1\}$ at line 11, then another process $p' \in P_{1 \oplus v}$ also outputs the opposite value $1 \oplus v$ at line 7.*

Proof. If some $p \in P_?$ outputs v at line 11, then it must have observed some OUTPUT$(1 \oplus v)$ at line 10. By C-Validity, this OUTPUT$(1 \oplus v)$ was communicated by some process $p' \in P_{1 \oplus v}$ at line 8, after it had output $1 \oplus v$ at line 7. ◀

Algorithm 1 Disagreement asynchronous algorithm for the Async case of lines 7-8 of Table 2, assuming $\frac{3}{2}t + 1 < n \geq 2$.

```
1  instantiation parameter: Boolean no_out ∈ {true, false}.
2  initialization: pick three subsets of processes P_0, P_1, P_? ⊂ P, |P_0| ≥ (t+1)/2,
       |P_1| ≥ t/2, |P_?| ≥ (t+1)/2, P_0 ∪ P_1 ∪ P_? = P, P_0 ∩ P_1 = P_1 ∩ P_? = P_0 ∩ P_? = ∅;
       pick some set of processes P_init ⊂ P, |P_init| ≥ t + 1.
3  code of every process p_init ∈ P_init is
4      if no_out = true and pseudo_random_pick({0,1}) = 0 then communicate INIT.
5  code of every process p_v ∈ P_v, v ∈ {0,1} is
6      if no_out = true then wait until p_v observed some INIT;
7      output v;
8      communicate OUTPUT(v).
9  code of every process p_? ∈ P_? is
10     wait until p_? observed some OUTPUT(v) for the first time;
11     output 1 ⊕ v.
```

▶ **Lemma 8.** *If a process $p \in P_0 \cup P_1$ outputs some value $v \in \{0,1\}$ at line 7, then all correct processes in $P_0 \cup P_1$ pass line 6.*

Proof. Let us assume that some process $p \in P_0 \cup P_1$ outputs a value $v \in \{0,1\}$. We now show that all correct processes in $P_0 \cup P_1$ pass line 6.

- If $no\_out =$ `false`, then this is immediate.
- If $no\_out =$ `true`, then all correct processes in $P_0 \cup P_1$ reach the wait statement at line 6. However, since p reached line 7, it must have passed that wait, thus observing some INIT. By C-Global-termination, all correct processes in $P_0 \cup P_1$ also observe INIT and therefore pass line 6. ◀

▶ **Lemma 9.** *If some process $p \in P$ outputs some value $v \in \{0,1\}$, then another process $p' \in P \setminus \{p\}$ also outputs the opposite value $1 \oplus v$.*

Proof. Let us assume that some process $p \in P$ outputs a value $v \in \{0,1\}$. If $p \in P_?$, then Lemma 7 states that the execution must produce the output set $\{0,1\}$ and we are done. Otherwise, if $p \in P_0 \cup P_1$, then Lemma 8 applies, and all correct processes in $P_0 \cup P_1$ pass line 6. We consider the following two opposite cases.

1. Case *(i)*: At least one correct $p' \in P_?$ exists. As all correct processes in $P_0 \cup P_1$ pass line 6, then, in particular, correct process $p_v^c \in P_0 \cup P_1$ (Observation 6) reaches line 8 and communicates some OUTPUT($\star$). By C-Local-termination and C-Global-termination, p' will eventually observe this OUTPUT($\star$) information, and will therefore pass the wait statement at line 10 (either after it observed the OUTPUT($\star$) communicated by p_v^c, or because it observed some prior OUTPUT($\star$)). Finally, p' will reach line 11 and output some value $v' \in \{0,1\}$. Lemma 7 therefore applies, and some other process $p'' \in P_{1 \oplus v'}$ also outputs the opposite value $1 \oplus v$.
2. Case *(ii)*: All processes in $P_?$ crash. In this case, at least $\lceil \frac{t+1}{2} \rceil$ crashes have already occurred on all processes of $P_?$, and the maximum number of remaining crashes is $t - \lceil \frac{t+1}{2} \rceil$.
   - If t is even, then $\lceil \frac{t+1}{2} \rceil = \frac{t+2}{2} = \frac{t}{2} + 1$ and $t - \lceil \frac{t+1}{2} \rceil = t - \frac{t}{2} - 1 = \frac{t}{2} - 1 < \frac{t}{2}$.
   - If t is odd, then $\lceil \frac{t+1}{2} \rceil = \frac{t+1}{2}$ and $t - \lceil \frac{t+1}{2} \rceil = t - \frac{t+1}{2} = \frac{t-1}{2} < \frac{t}{2}$.

Therefore, it is not possible to also crash all processes of P_0 or P_1, since they both contain at least $\frac{t}{2}$ processes. This implies that there are at least two correct processes $p_0 \in P_0, p_1 \in P_1$. Since all correct processes in $P_0 \cup P_1$ pass line 6 (Lemma 8), then p_0 and p_1 both reach line 7, and respectively output 0 and 1.

Therefore, in every case, if some value $v \in \{0,1\}$ is output by a process $p \in P$, the opposite value $1 \oplus v$ is also output by another process $p' \in P \setminus \{p\}$. ◀

▶ **Theorem 10.** *Under Async and* $\frac{3}{2}t + 1 < n \geq 2$*, Algorithm 1 instantiated with Boolean* $no\_out \in \{\texttt{true}, \texttt{false}\}$ *implements the following sets of output sets:*

$$O = \begin{cases} \{\{0,1\}\} & \text{if } no\_out = \texttt{false}, \\ \{\varnothing, \{0,1\}\} & \text{if } no\_out = \texttt{true}. \end{cases}$$

Proof. We first show that condition $\frac{3}{2}t + 1 < n \geq 2$ is sufficient to construct the sets of processes used in Algorithm 1, namely $P_0, P_1, P_?, P_{\text{init}}$.

- If t is even, then $\frac{3}{2}t$ is an integer and $n > \frac{3}{2}t + 1$ implies $n \geq \frac{3}{2}t + 2$. Also, $\frac{t}{2}$ is an integer. Hence, there are enough processes to set $|P_0| \geq \frac{t}{2} + 1$, $|P_1| \geq \frac{t}{2}$, and $|P_?| \geq \frac{t}{2} + 1$, thus satisfying the conditions of line 2. This implies that $|P_0| + |P_1| + |P_?| = \frac{3}{2}t + 2$, as desired.
- If t is odd, then $\frac{3t-1}{2}$ is an integer and $\frac{3}{2}t = \frac{3t-1}{2} + \frac{1}{2}$. Then $n > \frac{3}{2}t + 1$ implies $n \geq \frac{3t-1}{2} + 2$. Also, $\frac{t-1}{2}$ is an integer. Hence, there are enough processes to set $|P_0| \geq \frac{t-1}{2} + 1$, $|P_1| \geq \frac{t+1}{2}$, and $|P_?| \geq \frac{t-1}{2} + 1$, thus satisfying the conditions of line 2. This implies that $|P_0| + |P_1| + |P_?| = \frac{3t-1}{2} + 2$, as desired.

Furthermore, since we have $n > \frac{3}{2}t + 1 \geq t + 1$ and $|P_{\text{init}}| \geq t + 1$, then there are also enough processes to construct the set P_{init}. Note also that $n \geq 2$ is already implied by $n > \frac{3}{2}t + 1$. We consider the following two opposite cases.

- Case 1: $no\_out = \texttt{false}$. We must show $O = \{\{0,1\}\}$.
- Case 2: $no\_out = \texttt{true}$. We must show $O = \{\varnothing, \{0,1\}\}$.

We now prove that the following safety and completeness results hold for both Cases 1 and 2.

1. General-Safety: Impossibility of $\{0\}$ and $\{1\}$ in Cases 1 and 2. By corollary of Lemma 9, the output sets $\{0\}$ and $\{1\}$ cannot happen.
2. General-Completeness: Possibility of $\{0,1\}$ in Cases 1 and 2. Let us first show that there is an execution where the correct process $p_v^c \in P_0 \cup P_1$ (Observation 6) passes line 6.
   - In Case 1, we have $no\_out = \texttt{false}$, so p_v^c passes this line.
   - In Case 2, we have $no\_out = \texttt{true}$, so p_v^c reaches the wait statement at line 6. However, since we have $|P_{\text{init}}| \geq t + 1$, there is at least one correct process $p_{\text{init}}^c \in P_{\text{init}}$, and there exists an execution where p_{init}^c picks 0 and then communicates INIT at line 4, thus unlocking p_v^c at line 6.

   Therefore, there is an execution where p_v^c passes line 6, and then outputs some value $v \in \{0,1\}$ at line 7. By Lemma 9, another process also outputs the opposite value $1 \oplus v$, thus producing the output set $\{0,1\}$.

The safety of Case 2 follows from General-Safety. For Case 1, we also have to prove the impossibility of $\varnothing$. Since $no\_out = \texttt{false}$ in Case 1, correct process p_v^c (Observation 6) passes the condition at line 6 and outputs some $v \in \{0,1\}$ at line 7, and thus $\varnothing$ is impossible.

The completeness of Case 1 follows from General-Completeness. For Case 2, we also have to prove the possibility of $\varnothing$. Since $no\_out = \texttt{true}$ in Case 2, there is an execution where no process from P_{init} communicates INIT at line 4 (because all processes in P_{init} either crashed or picked 1 at line 4). In this execution, no process in $P_0 \cup P_1$ passes the wait statement at line 6, and no process in $P_?$ passes the wait instruction at line 10. Therefore, no process outputs any value in this execution, thus producing output set $\varnothing$. ◀

Algorithm 2 Disagreement synchronous algorithm for the Sync case of lines 7-8 of Table 2, assuming $t+2 \leq n \geq 2$.

1 **instantiation parameter:** Boolean $no\_out \in \{\texttt{true}, \texttt{false}\}$.
2 **initialization:** pick 2 distinct sequences of processes $S_0 = (p_1^0, ..., p_{\lfloor \frac{n}{2} \rfloor}^0) \in P^{\lfloor \frac{n}{2} \rfloor}$, $S_0 = (p_1^1, ..., p_{\lceil \frac{n}{2} \rceil}^1) \in P^{\lceil \frac{n}{2} \rceil}$ s.t $\forall p \in P : (p \in S_0 \wedge p \notin S_1) \vee (p \notin S_0 \wedge p \in S_1)$; pick some set of processes $P_{\text{init}} \subset P, |P_{\text{init}}| \geq t+1$.
3 **local variable of every process** $p_i^v \in S_v, v \in \{0,1\}$**:** value $v_{\text{chosen}} \leftarrow \perp$.
4 **code of every process** $p_{\text{init}} \in P_{\text{init}}$ **at synchronous round** $R = 1$ **is**
5 **if** $no\_out = \texttt{true}$ **and** pseudo_random_pick$(\{0,1\}) = 0$ **then** communicate INIT.
6 **code of every process** $p_v \in S_v, v \in \{0,1\}$ **at synchronous round** $R \in [1..\lceil \frac{n}{2} \rceil]$ **is**
Communication step
7 **if** $R = i+1$ **and** p_i^v has output some value **then** communicate OUTPUT(v_{chosen}).
Computation step
8 **if** $R = i$ **and** $(no\_out = \texttt{false}$ **or** p_i^v observed INIT$)$ **then**
9 $v_{\text{chosen}} \leftarrow 1 \oplus v$ **if** p_i^v observed some OUTPUT(v) **else** v;
10 output v_{chosen}.

6.2 Synchronous Disagreement Algorithm (Lines 7–8)

The algorithm of this section (Algorithm 2) addresses the Sync case of lines 7 and 8 in Table 2. It is instantiated by providing a Boolean $no\_out \in \{\texttt{true}, \texttt{false}\}$ (line 1), which determines if the empty $\varnothing$ output set can be produced or not.

At the initialization of the algorithm, all processes of P are divided into two totally-ordered sequences S_0 and S_1, of $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ processes respectively, such that $|S_0| + |S_1| = n$. Informally, the value $v \in \{0,1\}$ of a sequence S_v corresponds to the *default value* that the processes of S_v output if they did not observe that another process already output this value before. The index i of every process p_i^v in a sequence S_v determines the round numbers in which p_i^v will output a value (round i) and communicate information (round $i+1$). Moreover, a set of at least $t+1$ processes $P_{\text{init}} \subset P$ is also selected (which is only used when $no\_out = \texttt{true}$). Each process $p_i^v \in S_v$ also has a local variable v_{chosen}, initialized to the sentinel value $\perp$, which stores the output value of p_i^v between round i and $i+1$.

Every process $p_{\text{init}} \in P_{\text{init}}$ passes the condition at line 5 only if $\varnothing$ is not a forbidden output set ($no\_out = \texttt{true}$) and it pseudo-randomly picks 0. If it does, p_{init} communicate INIT. Every process $p_i^v \in S_v$ ($v \in \{0,1\}$) only performs actions during synchronous rounds i and $i+1$ (if $i < \lceil \frac{n}{2} \rceil$), because of the conditions at lines 7 and 8.

During synchronous round i, p_i^v does not pass the condition at line 7 during the communication step, but it can enter the condition at line 8, if $no\_out = \texttt{false}$ or if it observed some INIT. If it enters the condition, then it updates its v_{chosen} local variable: v_{chosen} stores the opposite of p_i^v's default value v if it observed that another process in the system already output its v, otherwise v_{chosen} stores v (line 9). Finally, p_i^v outputs v_{chosen} (line 10).

During synchronous round $i+1$ (if $i < \lceil \frac{n}{2} \rceil$), p_i^v passes the condition at line 7 if it has output a value in the previous round (i.e., $v_{\text{chosen}} \neq \perp$). If it does, it communicates its output value through some OUTPUT(v_{chosen}) information (line 7).

Before proving the correctness of Algorithm 2, we first show some intermediary results.

▶ **Lemma 11.** *There must be two distinct correct processes $p_i^v \in S_v, p_j^{v'} \in S_{v'}$ for $v, v' \in \{0,1\}$ and $i, j \in [1..\lceil \frac{n}{2} \rceil]$.*

Proof. Since $n \geq t+2$, then there are at least two correct processes $p, p' \in P$. Moreover, by definition, all processes $p'' \in P$ belong to S_0 or S_1, so p, p' belong to either S_0 or S_1, and their indices in these sequences are comprised between 1 and $\lceil \frac{n}{2} \rceil$. ◀

▶ **Lemma 12.** *If some process $p \in P$ outputs a value $w \in \{0,1\}$ at line 10, then two correct processes $p_i^v \in S_v, p_j^{v'} \in S_{v'}$ (where $i, j \in [1..\lceil \frac{n}{2} \rceil]$ and $v, v' \in \{0,1\}$) also output some values $v_1, v_2 \in \{0,1\}$ at line 10.*

Proof. Assume some process $p \in P$ outputs a value $w \in \{0,1\}$ at line 10 during synchronous round $k \in [1..\lceil \frac{n}{2} \rceil]$. Let us also consider the two correct processes $p_i^v, p_j^{v'}$ belonging to either S_0 or S_1 (Lemma 11), where $i, j \in [1..\lceil \frac{n}{2} \rceil]$ are round numbers and $v, v' \in \{0,1\}$ are default output values. Process p must have satisfied the condition at line 8, which leads to two opposite cases.

- If *no_out* = `false`, then $p_i^v, p_j^{v'}$ do the following during round number i and j, respectively: they enter the condition at line 8, reach line 10 and output some value in $\{0,1\}$.
- If *no_out* = `true`, then p has observed some INIT information, which, by C-Validity, must have been communicated during round 1 by some process $p_{\text{init}} \in P_{\text{init}}$ at line 5. Moreover, by C-Global-termination and C-Synchrony, all correct processes in P must also observe some INIT information by the end of round 1. In particular, correct processes $p_i^v, p_j^{v'}$ respectively satisfy the condition at line 8 at rounds i, j, reach line 10, and output some values $v_1, v_2 \in \{0,1\}$.

Therefore, $p_i^v, p_j^{v'}$ always output some values $v_1, v_2 \in \{0,1\}$ at line 10. ◀

▶ **Lemma 13.** *If some process $p_j^{v'} \in S_{v'}$ outputs the opposite of its default value $1 \oplus v' \in \{0,1\}$ at line 10 during round $j \in [2..\lceil \frac{n}{2} \rceil]$, then some other process $p_k^\star \in P$ must have output v' in a previous round $k < j$.*

Proof. If a process $p_j^{v'} \in S_{v'}$ outputs the opposite of its default value $1 \oplus v' \in \{0,1\}$ at line 10 during round $j \in [2..\lceil \frac{n}{2} \rceil]$, then it must have chosen $v_{\text{chosen}} = 1 \oplus v'$ at line 9, and therefore it must have observed some OUTPUT(v') information. By C-Validity, this OUTPUT(v') must have been communicated by some process $p_k^\star \in P$ at line 7 at the beginning of round $k+1$ (where $2 \leq k+1 \leq j$). Moreover, $p_k^\star$ must have output v' at line 10 during round k (where $1 \leq k < j$). ◀

▶ **Lemma 14.** *If some process $p \in P$ outputs a value $v \in \{0,1\}$ at line 10, then another process $p' \in P \setminus \{p\}$ also outputs the opposite value $1 \oplus v$ at line 10.*

Proof. Let us assume that some process $p \in P$ outputs a value $v \in \{0,1\}$ at line 10. By Lemma 12, two correct processes $p_i^v, p_j^{v'}$ belonging to either S_0 or S_1 respectively output some values $v_1, v_2 \in \{0,1\}$ at line 10, during synchronous rounds $i, j \in [1..\lceil \frac{n}{2} \rceil]$.

If either p_i^v or $p_j^{v'}$ output the opposite of their default value (i.e., if $v_1 = 1 \oplus v$ or $v_2 = 1 \oplus v'$), then Lemma 13 applies, and there is another process $p_k^\star$ that outputs v_3, which is the opposite value that they output (i.e., $v_3 = 1 \oplus v_1$ if $v_1 = 1 \oplus v$, or $v_3 = 1 \oplus v_2$ if $v_2 = 1 \oplus v'$), so we are done. Therefore, in the following, we assume that p_i^v and $p_j^{v'}$ output their default values, i.e., $v_1 = v$ and $v_2 = v'$. We consider the following two cases.

- Case *(i)*: $i = j$. In this case, p_i^v and $p_j^{v'}$ both output during the same synchronous round $i = j$. As p_i^v and $p_j^{v'}$ are distinct processes, then they cannot belong to the same sequence S_0 or S_1, and we must have $v \neq v'$. But since p_i^v and $p_j^{v'}$ respectively output v and v', then they output opposite values.

- Case *(ii)*: $i < j$ (without loss of generality). In this case, p_i^v has output before $p_j^{v'}$, $i \in [1..\lceil \frac{n}{2} \rceil - 1]$, and $j \in [2..\lceil \frac{n}{2} \rceil]$. Let us remark that correct process p_i^v must have communicated OUTPUT(v_1) at line 7 during round $i+1$ (where $2 \leq i+1 \leq j$), after it has ouput v_1 at line 10 during round i. By C-Local-termination, C-Global-termination, C-Synchrony, $p_j^{v'}$ must have observed OUTPUT(v_1) by the end of round $i+1 \leq j$. However, since $v_2 = v'$ (i.e., $p_j^{v'}$ outputs its default value v'), by line 9, $p_j^{v'}$ has not observed any OUTPUT(v') information during or before round $j \geq 2$. This necessarily means that $v_1 \neq v' = v_2$, and therefore that p_i^v and $p_j^{v'}$ output opposite values (i.e., $v_1 = 1 \oplus v_2$).

Whether in Case *(i)* or Case *(ii)*, processes p_i^v and $p_j^{v'}$ output opposite values at line 10 (i.e., $v_1 = 1 \oplus v_2$), which concludes the lemma. ◀

▶ **Theorem 15.** *Under Sync and $t+2 \leq n \geq 2$, Algorithm 2 implements the following sets of output sets:*

$$O = \begin{cases} \{\{0,1\}\} & \text{if } no\_out = \texttt{false}, \\ \{\varnothing, \{0,1\}\} & \text{if } no\_out = \texttt{true}. \end{cases}$$

Proof. We first show that condition $t+2 \leq n \geq 2$ is sufficient to construct the sequences and set of processes used in Algorithm 1: $S_0, S_1, P_{\text{init}}$. Since we have $n = \lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = |S_0| + |S_1|$, then there are enough processes to construct the sequences of distinct processes S_0, S_1. Furthermore, since we have $n \geq t+2$ and $|P_{\text{init}}| \geq t+1$, then there are enough processes to construct the set P_{init}. Let us also remark that $n \geq 2$ is already implied by $n \geq t+2$. We consider the following two opposite cases.

- In Case 1, we have $no\_out = \texttt{false}$. In this case, we must prove that the set of output sets is $O = \{\{0,1\}\}$.
- In Case 2, we have $no\_out = \texttt{true}$. In this case, we must prove that the set of output sets is $O = \{\varnothing, \{0,1\}\}$.

We now prove that the following safety and completeness results hold for both Cases 1 and 2.

1. General-Safety: Impossibility of $\{0\}$ and $\{1\}$ in Cases 1 and 2. By corollary of Lemma 14, the output sets $\{0\}$ and $\{1\}$ cannot happen.
2. General-Completeness: Possibility of $0, 1$ in Cases 1 and 2. By Lemma 11, there is a correct process p_i^v in S_0 or S_1 that reaches the condition at line 8 during synchronous round $i \in [1..\lceil \frac{n}{2} \rceil]$. Let us show that, no matter the case, there is an execution where p_i^v passes the condition at line 8.
   - In Case 1, we have $no\_out = \texttt{false}$, so p_i^v satisfies the condition line 8 at round i.
   - In Case 2, we have $no\_out = \texttt{true}$. Since we have $|P_{\text{init}}| \geq t+1$, there is at least one correct process $p_{\text{init}}^c \in P_{\text{init}}$, and there exists an execution where p_{init}^c picks 0 and then communicates INIT at line 2 during round 1. By C-Local-Termination, C-Global-termination, and C-Synchrony, p_i^v observes INIT by the end of round 1, and therefore, when it reaches round $i \geq 1$, the condition at line 8 is satisfied.

   Therefore, there is an execution where p_i^v passes line 8, and then outputs some value $v \in \{0,1\}$ at line 10 during round i. By Lemma 14, another process also outputs the opposite value $1 \oplus v$, thus producing the output set $\{0,1\}$.

The safety of Case 2 follows from General-Safety. For Case 1, we also have to prove the impossibility of $\varnothing$. By Lemma 11, there is a correct process p_i^v belonging to S_0 or S_1. As $no\_out = \texttt{true}$ in Case 2, then p_i^v must pass line 8 during round $i \in [1..\lceil \frac{n}{2} \rceil]$, reach line 10, and output some value $v' \in \{0,1\}$. Therefore, the output set $\varnothing$ is impossible.

The completeness of Case 1 follows from General-Completeness. For Case 2, we have to prove the possibility of $\varnothing$. Since $no\_out = \texttt{true}$ in Case 2, there is an execution where no process from P_{init} communicates INIT at line 2 (because all processes in P_{init} either crashed or picked 1 at line 2). In this execution, no process in P passes line 8, and therefore, no process outputs any value in this execution, thus producing output set $\varnothing$. ◀

7 Conclusion

In this work, we exhaustively characterize solvability conditions for binary-output tasks under crash failures. More particularly, we focus on the sets of distinct output values that executions of these tasks can produce, which breaks down binary-output tasks into 16 classes. Our results cover both necessity, via impossibility proofs, and sufficiency, via implementing algorithms, thereby offering a definitive picture (Table 2) of the boundary between possible and impossible. By abstracting away from specific input assumptions and focusing on possible outputs, the conditions we present also serve as lower bounds (albeit not necessarily tight) for any stronger problem formulation. Let us also remark that our necessary conditions apply to all types of tasks (whether colored or colorless [15]), while our sufficient conditions only apply to input-independent, binary-output, colorless tasks.

Some of the results connect back to some well-studied problems, such as binary consensus and symmetry breaking, as discussed in Section 4. One particularly interesting problem we discovered is the *disagreement* problem that requires the system to never "agree" on one single output value (0 or 1). To our knowledge, this particular problem has not been studied before, and it illustrates how our framework exposes subtle but fundamental problems that are not captured by classical formulations.

We propose in the following some possible extensions to our current framework. Currently, we do not require that all correct processes eventually output some value. It is left for future work to explore whether adding this requirement will change the conditions. Exploring the tight conditions in partial synchronous environments is another interesting venue. Observe that, in the cases where the tight conditions are the same in both synchrony and asynchrony, then they are also tight for partial synchrony. We leave the remaining cases (e.g., disagreement) to future investigation. Another logical extension would be to consider multi-valued outputs (i.e., not limited to 0/1), which will significantly increase the number of possible sets of output sets. Moreover, output sets hide some structural information of output values, such as their multiplicity or the information about which process outputs the value. This can be done by adding additional information to output sets or by directly studying output vectors. Finally, the task input and validity constraints (relation between input and output) are also aspects to be explored in future work.

References

1 Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires $t+1$ rounds. *Inf. Process. Lett.*, 71(3-4):155–158, 1999. doi:10.1016/S0020-0190(99)00100-3.

2 Timothé Albouy, Antonio Fernández Anta, Chryssis Georgiou, Mathieu Gestin, Nicolas Nicolaou, and Junlang Wang. AMECOS: a modular event-based framework for concurrent object specification. In *Proc. 28th Int'l Conference on Principles of Distributed Systems (OPODIS'24)*, volume 324 of *LIPIcs*, pages 4:1–4:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.OPODIS.2024.4.

3 Hagit Attiya and Ami Paz. Counting-based impossibility proofs for set agreement and renaming. *J. Parallel Distributed Comput.*, 87:1–12, 2016. doi:10.1016/J.JPDC.2015.09.002.

4 Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (Extended abstract). In *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, pages 27–30. ACM, 1983. doi:10.1145/800221.806707.

5 Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003. doi:10.1145/637437.637447.

6 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

7 Armando Castañeda, Damien Imbs, Sergio Rajsbaum, and Michel Raynal. Generalized symmetry breaking tasks and nondeterminism in concurrent objects. *SIAM J. Comput.*, 45(2):379–414, 2016. doi:10.1137/130936828.

8 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. The renaming problem in shared memory systems: An introduction. *Comput. Sci. Rev.*, 5(3):229–251, 2011. doi:10.1016/J.COSREV.2011.04.001.

9 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993. doi:10.1006/INCO.1993.1043.

10 Wei Chen. Abortable consensus and its application to probabilistic atomic broadcast. Technical report, Microsoft Research Asia, 2007. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2006-135.pdf.

11 Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *J. ACM*, 37(4):720–741, 1990. doi:10.1145/96559.96565.

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

13 Eli Gafni and Sergio Rajsbaum. Musical benches. In *Proc. 19th Int'l Conference on Distributed Computing (DISC'05)*, volume 3724 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2005. doi:10.1007/11561927_7.

14 Vassos Hadzilacos and Sam Toueg. Reliable broadcast and related problems. In Sape Mullender, editor, *Distributed Systems*, pages 97–145. Addison-Wesley, Reading, MA, 2nd edition, 1993. Chapter 5.

15 Maurice Herlihy, Dmitry N. Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann, 2013. URL: https://store.elsevier.com/product.jsp?isbn=9780124045781.

16 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999. doi:10.1145/331524.331529.

17 Damien Imbs, Sergio Rajsbaum, and Michel Raynal. The universe of symmetry breaking tasks. In *Proc. 18th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO'11)*, volume 6796 of *Lecture Notes in Computer Science*, pages 66–77. Springer, 2011. doi:10.1007/978-3-642-22212-2_7.

18 Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Morgan Kaufmann, 1997.

19 Ephraim Korach, Shay Kutten, and Shlomo Moran. A modular technique for the design of efficient distributed leader finding algorithms. *ACM Trans. Program. Lang. Syst.*, 12(1):84–101, 1990. doi:10.1145/77606.77610.

20 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

21 Achour Mostéfaoui, Sergio Rajsbaum, and Michel Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM*, 50(6):922–954, 2003. doi:10.1145/950620.950624.

22 Michel Raynal. Consensus in synchronous systems: A concise guided tour. In *Proc. 9th Pacific Rim Int'l Symposium on Dependable Computing (PRDC'02)*, pages 221–228. IEEE Computer Society, 2002. doi:10.1109/PRDC.2002.1185641.

23 Armando Castañeda Rojano. *A study of the wait-free solvability of weak symmetry breaking and renaming*. PhD thesis, Universidad Nacional Autónoma de México, 2010.

24 Nicola Santoro and Peter Widmayer. Time is not a healer. In *Proc. 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS'89)*, volume 349 of *Lecture Notes in Computer Science*, pages 304–313. Springer, 1989. doi:10.1007/BFB0028994.

25 Eugene Styer and Gary L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proce. 8th Annual ACM Symposium on Principles of Distributed Computing (PODC'89)*, pages 177–191. ACM, 1989. doi:10.1145/72981.72993.

26 Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009. doi:10.1145/1435417.1435432.

A Sufficiency: Additional Algorithms and Proofs

This appendix presents the remaining algorithms and correctness proofs used for proving the sufficiency of the tightness conditions of Table 2.

A.1 Communication-Less All-Output Algorithm (Lines 1–2, 11–15)

The algorithm of this section (Algorithm 3) addresses lines 1 to 2, and 11 to 15 in Table 2. It requires no communication, as it only involves one process outputting a pseudo-random value in isolation. Therefore, this algorithm works both in the Sync and Async timing models. This algorithm is instantiated with one parameter (line 1): a set of values $V \subseteq \{0, 1, \bot\}$ determining all output behaviors of the algorithm. Every process $p \in P$ only picks a pseudo-random value $v \in V$ (line 3) and outputs this value if it is not $\bot$ (line 4).

▶ **Theorem 16.** *Under any timing model (Async or Sync) and* $n \geq \max(\{|o| : o \in O\})$*, Algorithm 3 instantiated with the nonempty set of values* $V \subseteq \{0, 1, \bot\}$ *implements the following sets of output sets:*

$$O = \begin{cases} 2^V \setminus \{\varnothing\} & \text{if } \bot \notin V \text{ and } t < n, \\ 2^{(V \setminus \{\bot\})} & \text{if } \bot \in V \text{ and } t \leq n. \end{cases}$$

Proof. In the following lemma, we denote by $P_{\text{out}} \subseteq P$ the subset of processes that output a value at line 4. Let us proceed by exhaustion.

- In Case 1, we have $\bot \notin V$ and $t < n$. In this case, there is at least one correct process $p \in P$, and we must prove that Algorithm 3 implements the set of output sets $O = 2^V \setminus \{\varnothing\}$. Therefore, the set of forbidden output sets is $(2^{\{0,1\}} \setminus 2^V) \cup \{\varnothing\}$.
- In Case 2, we have $\bot \in V$ and $t \leq n$. In this case, all processes could crash, and we must prove that Algorithm 3 implements the set of output sets $O = 2^(V \setminus \{\bot\}))$. Therefore, the set of forbidden output sets is $2^{\{0,1\}} \setminus 2^{(V \setminus \{\bot\})}$.

We begin with general safety and completeness results that apply to both Cases 1 and 2.

1. General-Safety: Impossibility of all sets in $2^{\{0,1\}} \setminus 2^{(V \setminus \{\bot\})}$ in Case 1 and 2. Let us consider a set $o \in 2^{\{0,1\}} \setminus 2^{(V \setminus \{\bot\})}$. Since o is in $2^{\{0,1\}}$ but not in $2^{(V \setminus \{\bot\})}$, it means that o contains values that are not in $V \setminus \{\bot\}$. But as processes only output values in $V \setminus \{\bot\}$ at line 4, then o is impossible.
2. General-Completeness: Possibility of all sets in $2^{(V \setminus \{\bot\})} \setminus \{\varnothing\}$. Since $n \geq \max(\{|o| : o \in O\})$ and all failure patterns are possible, there is a set of executions where at least $\max(\{|o| : o \in O\})$ processes pass the condition at line 4 (either because $\bot \notin V$ or because all of these processes picked a non-$\bot$ value at line 3). In these executions, we have $|P_{\text{out}}| \geq \max(\{|o| : o \in O\})$. Among these executions, and for every $o \in 2^{(V \setminus \{\bot\})} \setminus \{\varnothing\}$, there is an execution where at least $|o|$ processes in $|P_{\text{out}}|$ pick and output each of the values in o at line 4, therefore producing output set o.

Algorithm 3 Communication-less symmetric algorithm for lines 1-2 and 11-15 of Table 2.

```
1 instantiation parameter: set of values V ⊆ {0, 1, ⊥}.
2 code of every process p ∈ P is
3     v ← pseudo_random_pick(V);
4     if v ≠ ⊥ then output v;
```

Algorithm 4 Communication-less algorithm for lines 9 and 10 of Table 2, with a single process outputting and assuming $n \geq 1$.

```
1 instantiation parameter: Boolean no_out ∈ {true, false}.
2 initialization: pick some p ∈ P.
3 code of every process p ∈ P is
4     V ← {0, 1, ⊥} if no_out = true else {0, 1};
5     v ≥ pseudo_random_pick(V);
6     if v ≠ ⊥ then output v.
```

The safety of Case 2 follows from General-Safety. For Case 1, we also need to show the impossibility of $\varnothing$. Since $\bot \notin V$, some process p is correct, must pass the condition at line 4, and output a value at line 4. Therefore $\varnothing$ is impossible.

The completeness of Case 1 follows from General-Completeness. For Case 2, we also need to show the possibility of $\varnothing$. There is an execution where $P_{\text{out}} = \varnothing$ (because all processes either crashed or picked $\bot$ at line 3), therefore producing output set $\varnothing$. ◀

A.2 Communication-Less Single-Output Algorithm (Lines 9-10)

The algorithm of this section (Algorithm 4) addresses lines 9 to 10 in Table 2. It requires no communication, as it only involves one process outputting a pseudo-random value in isolation. Therefore, this algorithm works both in the Sync and Async timing models.

This algorithm is instantiated with one parameter (line 1): a Boolean $no\_out \in \{\texttt{true}, \texttt{false}\}$, determining if the empty $\varnothing$ output set can be produced or not. At the initialization of the algorithm (line 2), one special process $p \in P$ is chosen to be the only one that performs actions (lines 3 and 6), while all other processes do nothing.

Process p first selects the set V corresponding to the different output behaviors it can follow (line 4): $\{0, 1\}$ if $no\_out = \texttt{false}$, or $\{0, 1, \bot\}$ if $no\_out = \texttt{true}$. Then, p pseudo-randomly picks a value v from V (line 5). If $v \neq \bot$, then p outputs v at line 6, otherwise p does nothing.

▶ **Theorem 17.** *Under any timing model (Async or Sync) and $n \geq 1$, Algorithm 4 instantiated with Boolean $no\_out \in \{\texttt{true}, \texttt{false}\}$ implements the following sets of output sets:*

$$O = \begin{cases} \{\{0\}, \{1\}\} & \text{if } no\_out = \texttt{false} \text{ and } t = 0, \\ \{\varnothing, \{0\}, \{1\}\} & \text{if } no\_out = \texttt{true} \text{ and } t \leq n. \end{cases}$$

Proof. Let us proceed by exhaustion.

- In Case 1, we have $no\_out = \texttt{false}$ and $t = 0$. In this case, no process can crash, and we must prove that Algorithm 4 produces the set of output sets $\{\{0\}, \{1\}\}$. Therefore, the set of forbidden output sets is $\{\varnothing, \{0, 1\}\}$.
- In Case 2, we have $no\_out = \texttt{true}$ and $t \leq 0$. In this case, all processes can crash, and we must prove that Algorithm 4 produces the set of output sets $\{\varnothing, \{0\}, \{1\}\}$. Therefore, the set of forbidden output sets is $\{\{0, 1\}\}$.

We begin with general safety and completeness results that apply to both Cases 1 and 2.

1. General-Safety: Impossibility of $\{0,1\}$ in Cases 1 and 2. For an algorithm to produce two outputs in some execution, there must be at least two processes in some execution that output. But by construction, in Algorithm 4, only process p can output a value at line 2.
2. General-Completeness: Possibility of $\{0\}$ and $\{1\}$ in Cases 1 and 2. Since $n \geq \max(\{|o| : o \in O\})$ and all failure patterns are possible, there must be some executions in which process p never crashes. Among these executions, there must be one in which p picks 0 (resp. 1) at line 5 and outputs 0 (resp. 1) at line 6, therefore producing the output set $\{0\}$ (resp. $\{1\}$).

The safety of Case 2 follows from General-Safety. For Case 1, we also need to show the impossibility of $\varnothing$. Since $no\_out = \texttt{false}$ and no process can crash, p will always pick some value at line 5 and output it at line 6.

The completeness of Case 1 follows from General-Completeness. For Case 2, we also need to show the possibility of $\varnothing$. There is some execution where p either crashes before outputting, or picks $\perp$ at line 5. In this execution, p will not reach line 6, therefore, producing the output set $\varnothing$. ◀

A.3 Timing-Adaptive Algorithm (Lines 3–6)

The algorithm of this section (Algorithm 5) addresses lines 3 to 6 in Table 2. It is timing-adaptive as it provides different control flows depending on whether the system is synchronous or asynchronous.

This algorithm is instantiated by providing two parameters (line 1): one value $v \in \{0,1\}$ and one Boolean $no\_out \in \{\texttt{true}, \texttt{false}\}$. Value v is the *default value*: it always has to be output by the system, as long as any value is output. Boolean no_out determines if the empty $\varnothing$ output set can be produced or not. At the initialization of the algorithm, one special process $p \in P$ is chosen (line 2) to be the one that will guarantee that the opposite value of v (i.e., $1 \oplus v$) can sometimes be output if v has also been output.

The code of every process $p' \in P$ that is not p is as follows (lines 3 and 6). Process p' passes the condition at line 4 only if $\varnothing$ is a forbidden output set or if p' pseudo-randomly flips the "correct" bit to 0. Then, p' outputs v (line 5) and communicates some OUTPUT(v) (line 6).

Algorithm 5 Asymmetric algorithm for lines 3-6 of Table 2, assuming $n \geq 2$. Moreover, for lines 3 and 5 (which allow the $\varnothing$ output set), it assumes $t \leq n$, and for lines 4 and 6 (which forbid the $\varnothing$ output set), it assumes $t < n$.

```
1  instantiation parameter: value v ∈ {0,1}, no_out ∈ {true, false}.
2  initialization: pick some process p ∈ P.
3  code of every process p' ∈ P \ {p} is
4      if no_out = false or pseudo_random_pick({0,1}) = 0 then
5          output v;
6          communicate OUTPUT(v).
7  code of every process p is
8      if no_out = false or pseudo_random_pick({0,1}) = 0 then
9          if the system is synchronous then wait for the communication step of round 1;
10         else if the system is asynchronous then wait for a predefined local time;
11         if p observed some OUTPUT(v) then output pseudo_random_pick({0,1});
12         else output v.
```

The code of process p is as follows (lines 7 and 12). Like with p', process p passes the condition at line 8 only if $\varnothing$ is a forbidden output set or if p pseudo-randomly flips the "correct" bit to 0. Then, p waits for a given time, depending on the timing model: if the system is synchronous, p waits for the communication step of synchronous round 1 (line 9), otherwise (i.e., if the system is asynchronous) it waits until some predefined local time[2] (line 10). After this wait, p outputs a random bit if it has observed some OUTPUT(v) (line 11), otherwise it outputs the default value v (line 12).

▶ **Theorem 18.** *Under any timing model (Async or Sync) and $n \geq 2$, Algorithm 5 instantiated with value $v \in \{0,1\}$ and Boolean no_out $\in \{\texttt{true}, \texttt{false}\}$ implements the following sets of output sets:*

$$O = \begin{cases} \{\{v\}, \{v, 1 \oplus v\}\} & \text{if } no\_out = \texttt{false} \text{ and } t < n, \\ \{\varnothing, \{v\}, \{v, 1 \oplus v\}\} & \text{if } no\_out = \texttt{true} \text{ and } t \leq n. \end{cases}$$

Proof. In the following lemma, we denote by $P_{\text{out}} \subseteq P$ the subset of processes that output a value at line 5, line 11, or line 12. Let us proceed by exhaustion.

- In Case 1, we have $no\_out = \texttt{false}$ and $t < n$. In this case, there is at least one correct process $p \in P$, and we must prove that Algorithm 5 produces the set of output sets $\{\{v\}, \{v, 1 \oplus v\}\} \setminus \{\varnothing\}$. Therefore, the set of forbidden output sets is $\{\varnothing, \{1 \oplus v\}\}$.
- In Case 2, we have $no\_out = \texttt{true}$ and $t \leq n$. In this case, all processes could crash, and we must prove that Algorithm 5 produces the set of output sets $\{\varnothing, \{v\}, \{v, 1 \oplus v\}\}$. Therefore, the set of forbidden output sets is just $\{1 \oplus v\}$.

We begin with general safety and completeness results that apply to both Cases 1 and 2.

1. General-Safety: Impossibility of $\{1 \oplus v\}$ in Cases 1 and 2. Any process $p' \in P \setminus \{p\}$ that outputs a value at line 5 will output v. Process p outputs value $1 \oplus v$ (with some probability) only if it has observed that at least one other process has output v (line 11). C-Validity ensures that some process p' first outputs v and then communicates this information (line 5 and line 6), and therefore that value v has also been output. Thus, $o = \{1 \oplus v\}$ is impossible.
2. General-Completeness: Possibility of all sets in $\{\{v\}, \{v, 1 \oplus v\}\}$ in Cases 1 and 2. We first argue the possibility of $\{v, 1 \oplus v\}$. Since $n \geq 2$ and all failure patterns (and communication-delay patterns in asynchrony) are possible, there is a set of executions where process p (the selected one) and at least another process $p' \in P \setminus \{p\}$ do not crash and output a value. That is, process p' passes the condition at line 4 and therefore outputs v, and process p passes the check at line 8 and observes OUTPUT(v) before reaching the condition at line 11, and therefore outputs $\{1 \oplus v\}$, with some non-null probability. Observe that any other process in $P_{\text{out}} \setminus \{p\}$ outputs v, yielding the output set $\{v, 1 \oplus v\}$. The possibility of $\{v\}$ follows similarly, for example, in executions that p crashes or also outputs v.

The safety of Case 2 follows from General-Safety. For Case 1, we also need to show the impossibility of $\varnothing$. Since $no\_out = \texttt{false}$ and $t < n$, at least one process must pass the check in line 4 or line 8 and thus output a value (either at line 5 for p', or line 11, line 12 for p). Therefore, $\varnothing$ is impossible.

The completeness of Case 1 follows from General-Completeness. For Case 2, we also need to show the possibility of $\varnothing$. Consider an execution where all processes have either crashed before outputting, or picked 1 at line 4 or line 8, yielding the output set $\varnothing$. ◀

[2] In this case, the assumption that, in asynchrony, every communication-delay pattern can happen, helps us to guarantee that p sometimes observes information before this local time, see Section 3.2.

Algorithm 6 Synchronous algorithm for the Sync case of line 10 of Table 2, assuming $t < n \geq 1$.

```
1 code of every process p ∈ P in synchronous round R = 1 is
    Communication step
2   v ← pseudo_random_pick({0, 1});
3   communicate PROPOSE(v);
    Computation step
4   if p observed some PROPOSE(0) then output 0;
5   else output 1.                          ▷i.e., p observed only PROPOSE(1)'s
```

A.4 Synchronous Binary Consensus Algorithm (Line 10)

The algorithm of this section (Algorithm 6) addresses the Sync case of line 10 in Table 2. Every process $p \in P$ starts by executing the *communication step* of the first synchronous round (lines 2 and 3). They first pick a pseudo-random value v from the set $\{0, 1\}$ at line 2, and communicate a PROPOSE(v) information at line 3. Then, every $p \in P$ moves to the *computation step* of the first synchronous round (lines 4 and 5). At this stage, we can show that all non-crashed processes see the same set of PROPOSE(v) information. Process p output 0 if it has observed any PROPOSE(0) information during round 1 (line 4), otherwise, p output 1 (line 5).

Let us remark that this algorithm only requires a single round of communicate/observe communication to terminate. To the familiar reader, this fact may seem incompatible with the well-known result stating that, in synchronous message-passing systems, consensus requires at least $t + 1$ communication rounds (where t is the maximum number of crashes) [1]. However, this apparent contradiction stems from the fact that, in a synchronous message-passing system, one round at the communicate/observe level may comprise up to $t + 1$ rounds at the send/receive level. This can be intuitively understood by considering that reliable broadcast, which is used to implement communicate/observe in message-passing networks, also requires $t + 1$ rounds to terminate [11]. This is, in particular, needed to guarantee that all processes correctly receive the sender's message despite the potential crashes.

▶ **Theorem 19.** *Under Sync and $0 \leq t < n \geq 1$, Algorithm 6 implements the set of output sets $O = \{\{0\}, \{1\}\}$.*

Proof. Under Sync and $0 \leq t < n \geq 1$, there is at least $n - t \geq 1$ correct processes. The safety of Algorithm 6 is proved in the following.

- Impossibility of $\varnothing$. Any correct process p necessarily outputs a value at line 4 or line 5, therefore $\varnothing$ is impossible.
- Impossibility of $\{0, 1\}$. Let us denote by $P_{\text{obs}} \subseteq P$ the set of processes that observe some OUTPUT(v) at the end of round 1. By C-Synchrony, all processes in P_{obs} must have observed the same set of PROPOSE(v) communicated at the beginning of round 1, and they must have therefore taken the same branch at line 4 or line 5, thus the output cannot contain 2 distinct values.

The completeness of Algorithm 6 is proved in the following.

- Possibility of $\{0\}$. There is an execution where a process p picks value $v = 0$ at line 2, communicates OUTPUT(0) at line 3, and observes its own OUTPUT(0) and outputs 0 at line 4. By the impossibility of $\{0, 1\}$, this execution will produce output set $\{0\}$.

- Possibility of $\{1\}$. Let us denote by $P_{\text{com}} \subseteq P$ the set of processes that communicate some OUTPUT(v) at the beginning of round 1. There is an execution where all processes in P_{com} have picked value $v = 1$ at line 2, and therefore communicate PROPOSE(1) at line 3. By C-Local-Termination, C-Global-Termination, and C-Synchrony, any process p must observe only PROPOSE(1) information at the end of round 1, and it must therefore output 1 at line 5. By the impossibility of $\{0, 1\}$, this execution will produce output set $\{1\}$. ◀