Generating Fast Indulgent Algorithms

Dan Alistarh, Seth Gilbert, Rachid Guerraoui, Corentin Travers

Generating Fast Indulgent Algorithms

Dan Alistarh*[1], Seth Gilbert[2], Rachid Guerraoui[1], and Corentin Travers**[3]

[1] EPFL, Switzerland
[2] National University of Singapore
[3] Université de Bordeaux 1/LaBRI, France

Abstract. Synchronous distributed algorithms are easier to design and prove correct than algorithms that tolerate asynchrony. Yet, in the real world, networks experience asynchrony and other timing anomalies. In this paper, we address the question of how to efficiently transform an algorithm that relies on synchronous timing into an algorithm that tolerates asynchronous executions. We introduce a transformation technique from synchronous algorithms to indulgent algorithms [13], which induces only a constant overhead in terms of time complexity in well-behaved executions.
Our technique is based on a new abstraction we call an asynchrony detector, which the participating processes implement collectively. The resulting transformation works for the class of *colorless* distributed tasks, including consensus and set agreement. Interestingly, we also show that our technique is relevant for *colored* tasks, by applying it to the renaming problem, to obtain the first indulgent renaming algorithm.

1 Introduction

The complexity of distributed tasks has been thoroughly studied both in the synchronous and asynchronous timing models. To better capture the properties of real-world systems, Dwork, Lynch, and Stockmeyer [11] proposed the *partially synchronous* model, in which the distributed system may have both synchronous and asynchronous periods, but eventually stabilizes and becomes synchronous. This line of research inspired the introduction of *indulgent* algorithms [13], i.e. algorithms that guarantee correctness and efficiency when the system is synchronous, and maintain safety even when the system is asynchronous. Several indulgent algorithms have been designed for specific distributed problems, such as consensus (e.g., [8, 17]). However, designing and proving correctness of such algorithms is usually a difficult task, especially if the algorithm has to provide good performance guarantees.

In this paper, we introduce a general transformation technique from synchronous algorithms to indulgent algorithms, which induces only a constant overhead in terms of time complexity. Our technique is based on a new primitive called an *asynchrony detector*, which identifies periods of asynchrony in a fault-prone asynchronous system. We showcase the resulting transformation to obtain *indulgent* algorithms for the important class of *colorless* agreement tasks, that includes consensus and set agreement. We also apply our transformation to the distinct class of *colored* tasks, to obtain the first indulgent algorithm for the renaming problem [4].

Central to our technique is a new abstraction, called an asynchrony detector, which we design as a distributed service for detecting periods of asynchrony. The service detects asynchrony both at a *local* level, by determining whether the view of a process is consistent with a synchronous execution, and at a *global* level, by determining whether the *collective* view of a set of processes could have been observed in a synchronous execution.
We present an implementation of an asynchrony detector, based on the idea that each process maintains a log of the messages sent and received, which it exchanges with other processes. This creates a view of the system for every process, which we use to detect asynchronous executions.

Based on this abstraction, we introduce a general technique allowing synchronous algorithms to tolerate asynchrony, while maintaining time efficiency in well-behaved executions. The main idea behind the transformation is the following: as long as the asynchrony detector signals a synchronous execution, processes run the synchronous algorithm. If the system is well behaved, then the synchronous algorithm yields an output, on which the process decides. Otherwise, if the detector notices asynchrony, we revert to an existing asynchronous backup algorithm with weaker termination and performance guarantees.

* Supported by the NCCR MICS Project
** Additional support from INRIA team REGAL and ANR project SPREADS

We first showcase the technique by transforming algorithms for agreement tasks, also called *colorless* tasks, which includes consensus and set agreement. Intuitively, a colorless task allows processes to adopt each other's output values without violating the task specification, while ensuring that every value returned has been proposed by a process.
We show that any synchronous algorithm solving a colorless task can be made indulgent at the cost of two extra rounds of communication. For example, if a synchronous algorithm solves synchronous consensus in $t + 1$ rounds, where t is the maximum number of crash failures (i.e. the algorithm is time-optimal), then the resulting indulgent algorithm will solve consensus in $t+3$ rounds if the system is initially synchronous, or will revert to a safe backup, e.g. Paxos [16,17] or ASAP [3], otherwise.
The crux of the technique is the hand-off procedure: we ensure that, if a process decides using the synchronous algorithm, any other process either decides or adopts a state which is consistent with the decision. In this second case, we show that a process can recover a consistent state by examining the views of other processes. The validity property will ensure that the backup protocol generates a valid output configuration.

We also apply our technique to the renaming problem [4], and obtain the first *indulgent* renaming algorithm. Starting from the synchronous protocol of [6], our protocol renames in a tight namespace of N names and decides in $\lceil \log N \rceil + 3$ rounds, in synchronous executions. Since the original synchronous algorithm terminated in $\lceil \log N \rceil + 1$ rounds, this suggests that the cost of tolerating asynchrony can be made constant for *colored* tasks as well. In asynchronous executions, the protocol renames in a namespace of size $N + t$, which is optimal [14].

Roadmap. In Section 2, we present the model, while Section 3 presents an overview of related work. We define asynchrony detectors in Section 4. Section 5 presents the transformation for colorless agreement tasks, while Section 6 applies it to the renaming problem. In Section 7 we discuss our results.

2 Preliminaries

We consider an eventually synchronous system with $N \geq 3$ processes $\Pi = \{p_1, p_2, \ldots, p_N\}$, in which $t < N/2$ processes may fail by crashing. A process that does not crash during an execution is *correct*. We say a process is *alive* (or *non-failed*) if it has not crashed up to the current point in the execution.

Processes communicate via message-passing in rounds, which we model much as in [2,8,9]. More precisely, time is divided into rounds, which are synchronous, i.e., each process has a local clock that signals the beginning and end of a round, and these clocks are synchronized. However, message delivery is asynchronous, i.e. there is no guarantee that a message sent by a process in a round is also delivered to the recipient in the same round. Each message is received at most once, no message is altered, and no message is received without being sent. We assume that a process always receives its own message in every round. Also, we assume that there exists a global stabilization time $GST \geq 0$ after which the system becomes synchronous, i.e. every message sent by a correct process during the execution is delivered, and every message sent after GST is received in the same round in which it was sent. We denote such a system by $\mathsf{ES(N, t)}$.

Although indulgent algorithms are designed to work in this partially synchronous setting, they are optimized for the case in which the system is initially *synchronous*, i.e. when $GST = 0$. We denote the synchronous message-passing model with $t < N$ failures by $\mathsf{S}(N, t)$. In case the system stabilizes at a later point in the execution, i.e. $0 < GST < \infty$, then the algorithms are still guaranteed to terminate, although they might be less efficient. If the system never stabilizes, i.e. $GST = \infty$, indulgent algorithms might not terminate, although they always maintain safety.

For simplicity, in this paper we consider *full-information* protocols, i.e. algorithms in which each process sends its whole state to all other processes, in every round. We note that our transformation technique will work for protocols that are not full information, as the process state needed by the transformation is maintained by the asynchrony detector. The algorithms we consider are deterministic. For simplicity, we assume that their proposal values are always integers.

Rounds and message delays. Note that, given this setup, we can ensure that every process receives at least $N - t$ messages that have been sent in *each* round, as follows. In each round, each process sends its message, then waits until the synchronous round timeout. If the number of messages received before the timeout is at least $N - t$, then the process performs local computation, and moves to the next round. Otherwise, the process waits until it receives at least

$N-t$ messages sent in the current round, and only then moves to the next round. Note that this waiting step may cause the process to become desynchronized with respect to the timeout for the current round. On the other hand, it ensures that every process receives at least $N-t$ messages for each round. Since the system is eventually synchronous, this ensures that no process waits forever for the $N-t$ messages sent in a round. The implementation of these extended rounds is described in detail in Section 4. Note that a process discards any message sent in a round R if it has already progressed to a later round $R' > R$.

Executions and Views. We define the *trace* of an execution α as a sequence of vectors $(REC^1, REC^2, \ldots)$ with the property that the vector REC^r associated with round r describes the set of messages received by each process in round r. In particular, if process p_i has not crashed by the end of round r, then the set $REC^r[i]$ contains the processes from which process p_i has received a message in round r. If p_i crashes during round r, then $REC^r[i] = \perp$. Notice that each set $REC^r[i] \neq \perp$ is of size at least $N-t$. Notice that a trace fully defines an execution.

In the following, we say that a round r is synchronous if 1) every non-failed process receives a message from each non-failed process in round r and 2) every message is received before the predefined synchronous timeout for the current round has elapsed. Similarly, we say that an execution is *synchronous* if every round in the execution is synchronous. Note that this implies that if process p_i receives a message m from process p_j in round $r \geq 2$, then every non-failed process received all messages sent by process p_j in all rounds $r' < r$.

Since the algorithms we consider are deterministic, the *view* (or *state*) of a process p at a round r is given by the messages that p received at round r and in all previous rounds. We say that two execution prefixes α and β are *indistinguishable* from the point of view of process p if p's state is the same at the end of α and at the end of β. We say that the *view* of process p at round r of an execution prefix α is *synchronous* if there exists an r-round synchronous execution prefix β which is indistinguishable from p's view at round r. Alternatively, we say that p's view is consistent with a synchronous execution.

Colorless and Colored Tasks. In the following, a task is a tuple $(\mathcal{I}, \mathcal{O}, \Delta)$, where $\mathcal{I}$ is the set of vectors of input values, $\mathcal{O}$ is a set of vectors of output values, and Δ is a total relation from $\mathcal{I}$ to $\mathcal{O}$. A solution to a task, given an input vector I, yields an output vector $O \in \mathcal{O}$ such that $O \in \Delta(I)$.

Intuitively, a *colorless* task is a terminating task in which any process can adopt any decision value of any other process, without violating the task specification, and in which any (decided) output value is a (proposed) input value. We also assume that the output values have to verify a predicate $\mathcal{P}$, such as agreement (all outputs are equal) or k-agreement (there are at most k distinct outputs). For example, in the case of consensus, the predicate $\mathcal{P}$ states that all output values should be equal. Let $val(V)$ be the set of values in a vector V. We precisely define this family of tasks as follows.

A colorless task satisfies the following properties: (1) Termination: every correct process eventually outputs; (2) Validity: for every $O \in \Delta(I)$, $val(O) \subseteq val(I)$; (3) The Colorless property: If $O \in \Delta(I)$, then for every I' with $val(I') \subseteq val(I) : I' \in \mathcal{I}$ and $\Delta(I') \subseteq \Delta(I)$. Also, for every O' with $val(O') \subseteq val(O) : O' \in \mathcal{O}$ and $O' \in \Delta(I)$. Finally, we assume that the outputs satisfy a generic property (4) Output Predicate: every $O \in \mathcal{O}$ satisfies a given predicate $\mathcal{P}$. Consensus and k-set agreement are canonical examples of colorless tasks.

By contrast, a *colored* task is a (terminating) task that is not colorless. Renaming [4] is the canonical example of colored task, since it requires that the names returned be distinct, which prevents a process from adopting another process's output.

Weak Termination and Quiescence. We use the following three notions of termination for a protocol. The first, called *strong* termination, or simply *termination*, means that after decision, each correct process stops executing the algorithm and returns. The second, weaker one, which we call *weak* termination, implies that each correct process will eventually *decide*, i.e. returns a decision value to the calling application, although the process may continue to run the protocol after the decision. Finally, the third termination condition is called *quiescence* [1], and means that eventually the protocol stops sending messages (although processes may continue to participate in the protocol).

The protocols resulting from our transformation only ensure *weak* termination and *quiescence*, and do not ensure strong termination. This is standard for indulgent and window of synchrony algorithms [2, 3, 9]. Intuitively, the reason is the following: since we combine two algorithms (a primary synchronous algorithm and a backup), the fact that a set of processes may decide using the first algorithm may imply that the set of correct participants to the backup algorithm

falls below a majority of processes, which might break the correctness of the backup. To avoid this scenario, we require processes to continue executing the algorithm even after they have decided, although in synchronous executions the processes are no longer required to take additional steps or to send new messages. For details, please see Sections 5 and 6.

3 Related Work

Starting with seminal work by Dwork, Lynch and Stockmeyer [11], a variety of different models have been introduced to express weak strenghtenings of the standard asynchronous model of computation. These include failure detectors [5], round-by-round fault detectors (RRFD) [12], and the GIRAF framework [15]. In particular, indulgent algorithms [13] have been introduced to capture the properties of algorithms that are fast when the system behaves well, and maintain safety otherwise.

In [8, 9], Guerraoui and Dutta address the complexity of indulgent consensus in the presence of an eventually perfect failure detector. They prove a tight lower bound of $t + 2$ rounds on the time complexity of the problem, even in synchronous runs, thus proving that there is an inherent price to tolerating asynchronous executions. Our approach is more general than that of this reference, since we transform a whole class of synchronous distributed algorithms, solving various tasks, into their indulgent counterparts. On the other hand, since our technique induces a delay of two rounds of communication over the synchronous algorithm, in the case of consensus, we miss the lower bound of $t + 2$ rounds by one round. This overhead comes from the generality of the approach, since the transformation also applies to k-set agreement and renaming.

Recent work studied the complexity of agreement problems, such as consensus [3] and k-set agreement [2], if the system becomes synchronous after an unknown stabilization time GST. In [3], the authors present a consensus algorithm that terminates in $f + 2$ rounds after GST, where f is the number of failures in the system. In [2], the authors consider k-set agreement in the same setting, proving that $\lfloor t/k \rfloor + 4$ rounds after GST are enough for k-set agreement, and that at least $\lfloor t/k \rfloor + 2$ rounds are required. The algorithms from these references work with the same time complexity in the indulgent setting, where $GST = 0$. On the other hand, algorithms generated using our transformation will not work in a window of synchrony that starts later than round one, since they are optimized to take advantage of the system being initially synchronous. From the point of view of the technique, references [2, 3] also use the idea of "detecting asynchrony" as part of the algorithms, although this technique has been generalized in the current work to address a large family of distributed tasks.

Reference [10] considered a setting in which failures stop after GST, in which case three rounds of communication are necessary and sufficient. Leader-based, Paxos-like algorithms, e.g. [16, 17], form another class of algorithms that tolerate asynchrony, and can also be seen as indulgent algorithms. Compared to our round model, the GIRAF framework [15] is more general, since it completely separates the algorithm's computation from its round waiting condition, as well as allowing the environment to provide additional "oracle output." Our technique assume that rounds are synchronized, although messages are not necessarily delivered in the same round in which they are sent.

A precise definition of colorless tasks is given in [7]. In this paper, we augment this previous definition to include the standard *validity* property, i.e. that a value decided by a colorless task has to be a value proposed.

4 Asynchrony Detectors

An asynchrony detector is a distributed service that detects periods of asynchrony in an asynchronous system that may be initially synchronous. The service returns a YES/NO indication at each process for every round, and has the property that processes which receive YES at some round share a synchronous execution prefix. Next, we make this definition precise.

Definition 1 (Asynchrony Detector). *Let d be a positive integer. A d-delay asynchrony detector in $\mathsf{ES}(\mathsf{N}, \mathsf{t})$ is a distributed service that eventually returns, for every round r, either YES or NO, at each process. The detector ensures the following properties.*

1. *(Local detection) If process p receives YES at round r, then there exists an r-round synchronous execution in which p has the same view as its current view at round r.*

2. *(Global detection) For all processes that receive* YES *in round r, there exists an $(r-d)$-round synchronous execution prefix $\mathcal{S}[1,2,\ldots,r-d]$ that is indistinguishable from their views at the end of round $r-d$.*
3. *(Non-triviality) The detector never returns* NO *during a synchronous execution.*

The *local detection* property ensures that, if the detector returns YES, then there exists a synchronous execution consistent with the process' view. On the other hand, the *global detection* property ensures that, for processes that receive YES from the detector, the $(r-d)$-round execution prefix was "synchronous enough", i.e. there exists a synchronous execution consistent with what these processes perceived during the prefix. The *non-triviality* property ensures that there are no false positives.

4.1 Implementing an Asynchrony Detector

Next, we present an implementation of a 2-delay asynchrony detector in $\mathsf{ES}(\mathsf{N},\mathsf{t})$, which we call $\mathsf{AD}(2)$. The pseudocode is presented in Figure 1.

The general structure of the detector is given in procedure detector. This procedure follows the structure of a generic distributed algorithm, that sends a message to all processes in every round, then receives all messages for the current round and processes the messages received to update the current state (see lines 4-11 of the detector procedure). Whenever used in conjunction with a (synchronous) distributed algorithm, the detector will piggy-back its messages on the messages sent by the protocol, to ensure that the detector notices any asynchrony that the algorithm encounters.

Asynchrony may be detected using two mechanisms. The first is triggered when the timeout for the current round has elapsed, i.e. $\mathsf{time}() > R_c.\mathsf{end}()$, and less than $N-t$ messages sent in the current round have been received by the process (line 6). Then the detector sets $synch_i$ to false, and will proceed to return a NO indication. For consistency, we require processes to wait until they receive $N-t$ messages that were sent in this round before they can move to the next round of detection (lines 5-7), although the execution prefix is already marked as asynchronous . (These $N-t$ messages will be useful in reconstructing a synchronous view, as described in Section 5.)

The second detection mechanism, implemented in the process procedure, ensures that processes maintain a detailed view of the state of the system by aggregating all messages received in every round. For each round R_c, each process maintains an $Active$ set of processes, i.e. processes that sent it a message in the round; all other processes are in the $Failed$ set for that round (lines 2–4). Whenever a process receives a new message, it merges the contents of the $Active$ and $Failed$ sets of the sender with its own (lines 9–10). Asynchrony is detected by checking if there exists any process that is in the $Active$ set in some round r, while being in the $Failed$ set in some previous round $r' < r$ (lines 11–13). In the next round, each process sends its updated view of the system together with a $synch$ flag, which was set to false, if asynchrony was detected.

4.2 Proof of Correctness

In this section, we prove that the protocol presented in the Section 4.1 satisfies the definition of an asynchrony detector. We will ensure that the three properties in the definition hold, and that the detector $\mathsf{AD}(2)$ eventually returns an indication for every round R, at every non-failed process.

First, it is trivial to check that, by definition, the implementation does not return false positives during a synchronous execution (condition 3). Next, for the proof of the remaining detector properties, notice that the timing detection mechanism in lines 5-7 of the detect procedure has no impact on the first two conditions, since a process receives YES from $\mathsf{AD}(2)$ only if all messages have been received before the round timeout. Therefore we can ignore this mechanism when proving that these conditions hold for our implementation.

To see that the *local* detection condition is satisfied, notice that the contents of the $Active$ and $Failed$ sets at each process p can be used to construct a synchronous execution which is coherent with process p's view.

In the following, we focus on proving the *global* detection property. We show that, for a fixed round $R > 0$, given a set of processes $P \subseteq \Pi$ that receive YES from $\mathsf{AD}(2)$ at the end of round $R+2$, there exists an R-round synchronous execution $\mathcal{S}[1,R]$ such that the views of processes in P at the end of round R are consistent with $\mathcal{S}[1,R]$.

We start the proof by showing a few simple properties of the $Active$ and $Failed$ sets, and then show how to build a synchronous execution based on the views of processes in P. We begin by proving that if two processes receive YES

```
1  procedure detector()_i
2    msg_i ← ⊥; synch_i ← true; Active_i ← [ ]; Failed_i ← [ ];
3    for each round R_c do
4      send( msg_i )
       /* wait until round timeout has elapsed and N − t messages have been received for the current round */
5      while |receive(R_c)| < (N − t) OR time < R_c.end() do
         /* if the timeout for the current round has elapsed, then the execution prefix is asynchronous */
6        if time > R_c.end() then synch_i ← false
7        wait()
       /* store the messages for the current round */
8      msgSet_i ← receive(R_c)
9      (synch_i, msg_i) ← process(msgSet_i, R_c)
10     if synch_i = true then output YES
11     else output NO

1  procedure process( msgSet_i, R_c )_i
2    if synch_i = true then
3      Active_i[R_c] ← processes from which p_i receives a message in round R_c
4      Failed_i[R_c] ← processes from which p_i did not receive a message in round R_c
5      if there exists p_j ∈ msgSet_i with synch_j = false then synch_i ← false
6      else
7        for every msg_j ∈ msgSet_i do
8          for round r from 1 to R_c do
9            Active_i[r] ← msg_j.Active_j[r] ∪ Active_i[r]
10           Failed_i[r] ← msg_j.Failed_j[r] ∪ Failed_i[r]
11       for round r from 1 to R_c − 1 do
12         for round k from r + 1 to R_c do
13           if (Active_i[k] ∩ Failed_i[r] ≠ ∅) then synch_i ← false
14   if synch_i = true then
15     msg_i ← (synch_i, (Active_i[r])_{r∈[1,R_c]}, (Failed_i[r])_{r∈[1,R_c]})
16   else msg_i ← (synch_i, ⊥, ⊥)
17   return (synch_i, msg_i)
```

Fig. 1. The AD(2) asynchrony detection protocol.

from the asynchronous detector in round $R + 2$, then they must have received each other's round $R + 1$ messages, either directly, or through a relay process. Note that, because of the round structure, a process's round $R + 1$ message only contains information that it has acquired up to round R.

In the following, we will use a superscript notation to denote the round at which the local variables are seen. For example, $Active_q^{R+2}[R + 1]$ denotes the set $Active[R + 1]$ at process q, as seen from the end of round $R + 2$.

Lemma 1. *Let p and q be two processes that receive* YES *from* AD(2) *at the end of round $R + 2$. Then $p \in Active_q^{R+2}[R + 1]$. Notice that, as a consequence, for every round $r \leq R$, $Active_p^R[r] \subseteq Active_q^{R+2}[r]$, and $Failed_p^R[r] \subseteq Failed_q^{R+2}[r]$.*

Proof. Assume, for the sake of contradiction, that $p \notin Active_q^{R+2}[R + 1]$. Then, by lines 9–10 of the process() procedure, none of the processes that send a message to q in round $R + 2$ received a message from p in round $R + 1$. However, this set of processes contains at least $N - t > t$ elements, and therefore, in round $R + 2$, process p receives a message from at least one process that did not receive a message from p in round $R + 1$. Therefore $p \in Active_p^{R+2}[R + 2] \cap Failed_p^{R+2}[R + 1]$ (recall that p receives its own message in every round). Following the process() procedure for p, we obtain that $synch_p =$ false in round $R + 2$, which means that process p receives NO from AD(2) in round $R + 2$, contradiction.

Therefore, we have shown that process q must receive process p's round $R+1$ message, either directly or through a relay. The second statement in the Lemma follows by lines 7-10 of the detector pseudocode. □

Next, we notice that the sets $Active$ and $Failed$ that processes in P see are consistent up to the end of round R. (Recall that P is a set of processes that received YES from AD(2) at the end of round $R+2$.)

Lemma 2. *Let p and q be two processes in P as defined above. Then, for all rounds $k < l \leq R$, $Active_p^R[l] \cap Failed_q^R[k] = \emptyset$, where the Active and Failed sets are seen from the end of round R.*

Proof. We prove that, given $R \geq l > k$, $Active_p^R[l] \cap Failed_q^R[k] = \emptyset$. Assume, for the sake of contradiction, that there exist rounds $k < l \leq R$ and a process s such that $s \in Active_p^R[l] \cap Failed_q^R[k]$. Lemma 1 ensures that, since p and q communicate in round $R+1$ (either directly or through a relay process), it holds that $s \in Failed_p^{R+2}[k]$. This means that $s \in Active_p^{R+2}[l] \cap Failed_p^{R+2}[k]$, for $k < l$, therefore p cannot receive YES in round $R+2$, contradiction. □

The next lemma provides a sufficient condition for a set of processes to share a synchronous execution up to the end of some round R. The proof follows from the observation that the required synchronous execution $\mathcal{E}$ can be constructed by exactly following the contents of the *Active* and *Failed* sets by processes at every round in the execution.

Lemma 3. *Let $\mathcal{E}$ be an R-round execution prefix in $\mathsf{ES}(\mathsf{N},\mathsf{t})$, and P be a set of processes in Π such that, at the end of round R, the following properties are satisfied:*

1. *For any p and q in P, not necessarily distinct, and any round $r \in \{1,2,\ldots,R-1\}$, we have that $Active_p^R[r+1] \cap Failed_q^R[r] = \emptyset$.*
2. *$|\bigcap_{p \in P} Active_p^R[R]| \geq N - t$.*
3. *For any p and q in P, $p \in Active_q^R[R]$.*

Then there exists a synchronous *execution prefix $\mathcal{E}'$ which is indistinguishable from the views of processes in P at the end of round R.*

Proof. Recall that an execution is defined by a sequence of vectors $(REC^1, REC^2, \ldots)$, where $REC^r[p]$ contains the set of messages received by process p in round r. We will build a synchronous execution $\mathcal{E}'$ based on the sets $Active_p^r[r]$, for each process p and round r, which are the sets of processes from which p has received messages in round r of $\mathcal{E}$.

We build the execution trace $\mathcal{E}'$ in a top-down manner, starting from the last round R. In this round, only processes in $S^R := P \cup \bigcap_{p \in P} Active_p^R[R]$ receive messages, while the rest are assumed to be crashed in $\mathcal{E}'$. Notice that, in fact, by condition 3, $P \subseteq \bigcap_{p \in P} Active_p^R[R]$, which means that $S^R = \bigcap_{p \in P} Active_p^R[R]$. For each process $p \in S^R$, we define the set of messages it receives in $\mathcal{E}'$ as follows. If $p \in P$, then $REC^R[p] = Active_p^R[R]$. Otherwise, if $p \in \bigcap_{p \in P} Active_p^R[R] \setminus P$, then we set $REC^R[p] = \bigcap_{p \in P} Active_p^R[R] \cup P$. Notice that processes in $S^R \setminus P$ may be crashed in round R of $\mathcal{E}$, but we simulate them as alive in $\mathcal{E}'$.

We now need to define the set of messages that each process in S^R received in round $R-1$ of $\mathcal{E}'$. For each process p in S^R, we define $REC^{R-1}[p]$ to be simply $Active_p^{R-1}[R-1]$, i.e. the set of messages that p received in round $R-1$ of execution $\mathcal{E}$. We work by backward induction on the round number $1 \leq r \leq R-1$ to define the set S^r as $\bigcap_{p \in S^{r+1}} Active_p^r[r]$. For each process $q \in S^r$, the set of messages it received in round $r-1$ is $Active_q^{r-1}[r-1]$.

This defines an execution trace $\mathcal{E}'$. First, notice that the processes in P have the same state at the end of $\mathcal{E}$ as at the end of $\mathcal{E}'$ since they received the exactly the same sets of messages in the two execution prefixes. We now check that the execution $\mathcal{E}'$ we have built is synchronous.

First, we prove that $S^R \subseteq S^{R-1} \subseteq \ldots S^1$. We proceed by backward induction on the round number $r \in \{R, R-1, \ldots, 1\}$. For the base case, we need to show that $S^R \subseteq S^{R-1}$. Assume for the sake of contradiction that there exists a process $s \in S^R \setminus S^{R-1}$. Then there exists a process q in S^R that did not receive a message from s in round $R-1$, although every process in P received a message from s in round R, i.e. for every $p \in P$, we have that $s \in Active_p^R[R]$. However, by condition 3 combined with the definition of S^R, the fact that $q \in S^R$ implies that every process in P receives q's message in round R, which implies that, for every process $p \in P$, $s \in Failed_p^R[R-1]$. However, this

implies that $s \in Active_p^R[R] \cap Failed_p^R[R-1]$, contradicting condition 1 of the Lemma. We therefore obtain that the base case holds.

For the induction step, we have that, for a fixed $1 \leq k \leq R-2$, it holds that $S^R \subseteq S^{R-1} \subseteq \ldots \subseteq S^{R-k}$, and we need to show that $S^{R-k} \subseteq S^{R-k-1}$. Notice that, in particular, the induction assumption implies that the set P is included in S^{R-k}. Assume for the sake of contradiction that there exists a process $s \in S^{R-k} \setminus S^{R-k-1}$. Then, by the definition of S^{R-k-1}, there must exist a process $q \in S^{R-k}$ such that $s \in Failed_q^{R-k-1}[R-k-1]$. However, since $q \in S^{R-k}$ and $P \subseteq S^{R-k+1}$, we get that every process p in P receives q's message in round $R-k$, implying that $s \in Failed_p^R[R-k-1]$ for any $p \in P$. On the other hand, since $s \in S^{R-k}$ and $P \subseteq S^{R-k+1}$, we have that, for any $p \in P$, $s \in Active_p^{R-k}[R-k]$. We therefore obtain that there exists a process $p \in P$ such that $Active_p^R[R-k] \cap Failed_p^R[R-k-1] \neq \emptyset$, contradicting condition 1 of the Lemma.

The previous argument implies that $S^R \subseteq S^{R-1} \subseteq \ldots S^1$. Next, we check that each process receives at least $N-t$ messages in each round $1 \leq r \leq R$. For rounds $r < R$, this follows from the properties of the original execution $\mathcal{E}$. For round R, this follows from property 2 in the Lemma statement.

Finally, we prove that no process is falsely suspected to have failed in some round of the execution $\mathcal{E}'$ we have built. Specifically, we show that in $\mathcal{E}'$ it cannot happen that a message from a process p is received in a round r_2 after process p has been placed in the $Failed$ set for a round r_1 with $r_1 < r_2$. Assuming for the sake of contradiction that this can occur for rounds $r_1 < r_2 < R$, by the structure of the execution $\mathcal{E}'$, we obtain that there exist processes q_1 and q_2 in $P \cup \bigcap_{p \in P} Active_p^R[R]$ such that $p \in Active_{q_1}^{R-1}[r_2]$ and $p \in Failed_{q_2}^{R-1}[r_1]$. Since every process in $P \cup \bigcap_{p \in P} Active_p^R[R]$ sends a message to a process in P in round R, it then follows that there exist processes p_1 and p_2 in P such that $p \in Active_{p_1}^R[r_2]$ and $p \in Failed_{p_2}^R[r_1]$ in the original execution $\mathcal{E}$. This contradicts condition 1 in the statement of the Lemma.

Finally, we prove that synchrony cannot be broken in the last round R. First, notice that condition 3 and the fact that we allow only processes in $\bigcap_{p \in P} Active_p^R[R]$ to finish round R ensure that no process that is alive at the end of $\mathcal{E}'$ is considered failed by any other process in P.

The only case left is if a process q_1 that is alive in round R, i.e. $q_1 \in S^R := P \cup \bigcap_{p \in P} Active_p^R[R]$, receives in R a message from a process p that is in $Failed_{q_2}^R[r_1]$, for some process $q_2 \in S^R$ and some round $r_1 < R$. Notice that process q_2 cannot be in P, since it would contradict condition 1. Therefore, process q_2 is in $\bigcap_{p \in P} Active_p^R[R] \setminus P$, so in our construction of $\mathcal{E}'$ it is simulated as receiving messages from processes in S^R in round R. Since q_2 has $p \in Failed_{q_2}^R[r_1]$, there must exist a process q that sends a message to q_2 in round R such that $p \in Failed_q^{R-1}[r_1]$ (the process q could be q_2 itself). Notice that $q \notin P$, since otherwise condition 1 would be contradicted. Then q is in P, which again contradicts condition 1 since $Failed_q^R[r_1] \cap Active_{q_1}^R[R] \neq \emptyset$.

We can therefore conclude that the trace obtained describes a synchronous execution $\mathcal{E}'$ which is indistinguishable from $\mathcal{E}$ from the point of view of processes in P. □

We now prove that if a set of processes P receive YES from $\mathsf{AD}(2)$ at the end of some round $R+2$, then there exists a synchronous execution consistent with their views at the end of round R, for any $R > 0$.

Lemma 4. *Let $R > 0$ be a round and P be a set of processes that receive YES from $\mathsf{AD}(2)$ at the end of round $R+2$. Then there exists a synchronous execution consistent with their views at the end of round R.*

Proof. We show that the views of processes in P at the end of round R have to verify the conditions in Lemma 3, which ensures the existence of the desired synchronous execution.

Condition 1 holds by Lemma 2. For condition 2, assume for the sake of contradiction that $|\bigcap_{p \in P} Active_p^R[R]| < N-t$. Therefore there exist at most $N-t-1$ common processes from which all processes in P receive messages in round R. We show that, intuitively, there will not be enough processes left from which processes in P may receive messages from in round $R+2$ while still all receiving YES from the $\mathsf{AD}(2)$.

More formally, in round $R+2$, for each process p in P receiving the set M_p of messages with $|M_p| \geq N-t$, there exists another process q in P such that one process $s \in M_p$ was seen as failed by q in round R, i.e. $s \in Failed_q^R[R]$. However, by Lemma 1, $q \in Active_p^{R+2}[R+1]$, therefore, by the structure of the algorithm, $s \in Active_p^{R+2}[R+2] \cap Failed_p^{R+2}[R]$, a contradiction with the fact that s receives YES from the asynchrony detector in round $R+2$. Therefore $|\bigcap_{p \in P} Active_p^R[R]| \geq N-t$.

Finally, we derive condition 3 from Lemma 1. This result ensures that, for every two processes p and q in P, $p \in Active_q^{R+2}[R+1]$, and $q \in Active_p^{R+2}[R+1]$. Assume for contradiction that $p \notin Active_q^R[R]$, i.e. that q does not directly receive p's message in round R. By definition, this implies that $p \in Failed_q^R[R]$; therefore, $p \in Failed_q^{R+2}[R] \cap Active_q^{R+2}[R+1]$, contradicting the fact that q receives YES from AD(2) in round $R+2$. The converse claim follows symmetrically.

Therefore, all the conditions of Lemma 3 hold, so there exists a synchronous execution $\mathcal{E}'$ which is consistent with the views of each process in P up to the end of round R. □

Returning to the proof, we have obtained so far that conditions 1) through 3) of the asynchrony detector definition hold for the AD(2) protocol. Finally, we need to prove that, for every round R, the detector returns an indication at every non-failed process.

Lemma 5 (Termination). *The* AD(2) *implementation ensures that, eventually, every non-failed process obtains a* YES/NO *indication for every round R.*

Proof. For the sake of contradiction, consider a non-failed process p that does not obtain an indication for a given round R. Without loss of generality, let R be the first such round in the execution. By examination of the code, this may only occur if p is stuck waiting on line 5 of the protocol and never receives the $N-t$ required messages for round R. However, since there exist at least $N-t$ correct processes, each of these processes eventually sends a message for round R in this execution (since R is the first round in which a process gets stuck). Since the system is eventually synchronous, all these $N-t$ messages eventually get delivered, therefore process p eventually progresses past line 5 of the protocol, contradicting our assumption on R. □

Therefore, we have verified all the properties of an asynchrony detector.

Theorem 1. *The* AD(2) *algorithm in Figure 1 is a correct implementation of an asynchrony detector.*

5 Generating Indulgent Algorithms for Colorless Tasks

5.1 Transformation Description

We present an emulation technique that generates an indulgent protocol in ES(N,t) out of *any* protocol in S(N,t) solving a given colorless task T, at the cost of two extra communication rounds for decision. If the system is not synchronous, the generated protocol will run a given backup protocol Backup which ensures safety, even in asynchronous executions. For example, if a protocol solves synchronous consensus in $t+1$ rounds (i.e. it is optimal), then the resulting protocol will solve consensus in $t+3$ rounds if the system is initially synchronous. Otherwise, the protocol reverts to a safe backup, such as Paxos [16], or ASAP [3].

We fix a protocol $\mathcal{A}$ solving a colorless task in the synchronous model S(N,t). The running time of the synchronous protocol is known to be of R rounds. In the first phase of the transformation, each process p runs the AD(2) asynchrony detector in parallel with the protocol $\mathcal{A}$, as long as the detector returns a YES indication at every round. Note that the protocol's messages are included in the detector's messages (or vice-versa), preventing the possibility that the protocol encounters asynchronous message deliveries without the detector noticing. If the detector returns NO during this first phase, the process stops running the synchronous protocol, and continues running only AD(2). If the process receives YES at the end of round $R+2$, then it returns the decision value that $\mathcal{A}$ produced at the end of round R. Note that, since AD(2) returns YES at process p at the end of round $R+2$, it follows that it must have returned YES at p at the end of round R as well. The local detection property of the asynchrony detector implies that the protocol $\mathcal{A}$ has to return a decision value, since it executes a synchronous execution. The decision value is then returned to the calling application.

After deciding, the process executes a backup algorithm such as Paxos [16], or ASAP [3], if the task is consensus, or K4 [2], if the task is k-set agreement. Importantly, note that the process runs this backup algorithm starting in *decided* state, keeping its current decision. For all the backup algorithms we consider [2, 3, 16], this implies that the process will not send any message or take any steps as part of the algorithm, unless it receives a message from a process that has not yet decided.

On the other hand, if the process receives NO from $\mathsf{AD}(2)$ in round $R+2$, i.e. asynchrony was detected, then the process needs to run the second phase of the transformation. In phase two, the process will run a backup agreement protocol that tolerates periods of asynchrony, e.g. [2,3,16]. The main question is how to initialize the backup protocol, given that some of the processes may have already decided in phase one, without breaking the properties of the task. We solve this problem as follows.

Let p be a process in this situation and let $Supp$ (the *support* set) be the set of processes that received YES from $\mathsf{AD}(2)$ in round $R+1$ that process p receives messages from in round $R+2$. There are two cases. (1) If the set $Supp$ is empty, then the process starts running the backup protocol using its initial proposal value. (2) If the set $Supp$ is non-empty, then the process obtains a new proposal value as follows. It picks one process q from $Supp$ and adopts its state at the end of round $R-1$. Then, in round R, it simulates process q receiving the messages received by *every* process j in the support set in round R. More precisely, the process simulates the state of process q after receiving the messages in $\bigcap_{j \in Supp} msgSet_j^{R+1}[R]$ in round R, where we maintain the notation used in Section 4. We say that in this case process p *adopts* a view. We will show that in this case, the simulated protocol $\mathcal{A}$ will necessarily return a decision value at the end of simulated round R. The process p then runs the backup protocol, using as initial value the decision value resulting from the simulation of the first R rounds.

5.2 Proof of Correctness

We now prove that the resulting protocol verifies the task specification. The proofs of validity and the colorless property follow immediately from the properties of the $\mathcal{A}$ and Backup protocols, therefore we will concentrate on proving that the resulting protocol also satisfies the output predicate $\mathcal{P}$, and that the transformation satisfies weak termination and quiescence after $R+2$ rounds in synchronous executions. We begin by proving that the output predicate $\mathcal{P}$ holds.

Lemma 6 (Output Predicate). *The indulgent transformation protocol satisfies the output predicate $\mathcal{P}$ associated to the task T.*

Proof. Assume for the sake of contradiction that there exists an execution in which the output of the transformation breaks the output predicate $\mathcal{P}$. If all process decisions are made at the end of round $R+2$, then, by the global detection property of $\mathsf{AD}(2)$, there exists a synchronous execution of $\mathcal{A}$ in which the same outputs are decided, and these outputs break the predicate $\mathcal{P}$, a contradiction with the correctness of $\mathcal{A}$. If all decisions occur after round $R+2$, first notice that, by the validity and colorless properties, the inputs processes propose to the Backup protocol are always valid inputs for the task. It follows that, since all decisions are output in an execution of the Backup protocol, there exists an execution of the Backup protocol in which the predicate $\mathcal{P}$ is broken, again a contradiction.

Therefore, at least one process outputs at the end of round $R+2$, and some processes decide at some later round. We prove the following claim.

Claim. Given a process d that decides at the end of round $R+2$, let $Q = \{q_1, \ldots, q_\ell\}$ be the set of alive (non-crashed) processes at the end of round $R+2$. Then (i) all processes in Q will have a non-empty support set $Supp$ and (ii) there exists an R-round synchronous execution consistent with the views that processes in Q adopt at the end of round $R+2$.

Proof. First, let d be a process that decides at the end of round $R+2$. Then, in round $R+2$, process d received a message from at least $N-t$ processes that got YES from $\mathsf{AD}(2)$ at the end of round $R+1$. (In fact, all processes that d receives messages from in this round had received a YES indication.) Since $N \geq 2t+1$, it follows that every process that has not crashed by the end of round $R+2$ will have received at least one message from a process that has received YES from $\mathsf{AD}(2)$ in round $R+1$. Hence, all non-crashed processes that get NO from $\mathsf{AD}(2)$ in round $R+2$ will adopt a view, which ensures the first claim.

Recall that the set $Q = \{q_1, \ldots, q_\ell\}$ denotes the non-crashed processes at the end of round $R+2$. By the above claim, we know that these processes either decide or simulate an execution. We prove that all views simulated in this round are consistent with a synchronous execution up to the end of round R, in the sense of Lemma 3.

First, we prove that the intersection of their simulated views in round R contains at least $N-t$ messages. Without loss of generality, we assume that d is the only deciding process in this round, and all the other processes have to

adopt views. (For the set of processes that decide at the end of round $R+2$, the existence of the execution follows by Lemma 4.)

We first notice that the processes from which d receives messages in round $R+2$ are necessarily in this intersection. Assume for contradiction that there is a process s from which d receives a message in round $R+2$, and a process q that receives YES from the asynchrony detector at the end of round $R+1$ such that $s \notin Active_q^{R+1}[R]$. Then, it follows that *none* of the processes that q receives a message from in round $R+1$, which we denote by the set M_q, have $s \in Active^R[R]$–that is, none of them received a message from s in round R. This implies that $s \in Failed_i^R[R]$ for every process $i \in M_q$. Since $|M_q| \geq N - t$, it follows that the deciding process d receives a message from a process in M_q in round $R+1$, therefore $s \in Failed_d^{R+2}[R]$. On the other hand, the process s is in $Active_d^{R+2}[R+2]$, which contradicts the fact that d receives a YES indication from the detector. This proves the second condition of Lemma 3.

To prove the first condition of Lemma 3, note that, by the above argument, process d's view of round R, i.e. the set $Active_d^{R+2}[R]$, contains all messages simulated as received in round R by the processes that receive NO in round $R+2$. More precisely, for each process p that adopts a view at the end of round $R+2$, the set of messages $msgSet_p[R]$ it simulates receiving in round R is included in $Active_d^{R+2}[R]$. Assume for the sake of contradiction that there exist processes p_1 and p_2 that adopt views at $R+2$ and rounds $1 \leq r_1 < r_2 \leq R$ such that $Failed_{p_1}^R[r_1] \cap Active_{p_2}^R[r_2] \neq \emptyset$. Let s be a process in this non-empty intersection.

We show that s is necessarily in $Failed_d^{R+2}[r_1] \cap Active_d^{R+2}[r_2]$, generating a contradiction. For this, notice that, in round R, process p_1 receives a message from a process that has s in its $Failed$ set for round r_1 at the end of $R-1$. Since $msgSet_{p_1}^{R+2}[R]$ is included in $Active_d^{R+2}[R]$, it follows that $s \in Failed_d^{R+2}[r_1]$. Similarly, we consider process p_2 and obtain that $s \in Active_d^{R+2}[r_2]$. Since $r_1 < r_2$, we obtain a contradiction with the fact that d receives YES from the detector and decides at the end of round $R+2$.

Finally, we need to show that given any two processes p_1 and p_2 that either adopt a view or decide at the end of round $R+2$, we have that $p_1 \in msgSet_{p_2}^R[R]$ and $p_2 \in msgSet_{p_1}^R[R]$. If p_1 and p_2 both decide at $R+2$, the claim follows easily from Lemma 4. If p_1 and p_2 both adopt views from processes q_1 and q_2 in the support set, respectively, the claim follows from Lemma 1, since both q_1 and q_2 receive YES from the detector at the end of round $R+1$. Finally, if p_1 adopts a view from q_1 and p_2 decides, then the claim follows from Lemma 4, applied to processes q_1 and p_2 at the end of round $R+1$.

Therefore, the three necessary conditions hold, and we can apply Lemma 3 to obtain that there exists a synchronous execution of the protocol $\mathcal{A}$ in which the processes in Q obtain the same decision values as the values obtained through the simulation or decision at the end of round $R+2$. □

Returning to the proof of the output predicate Lemma, recall that we assumed there exists process d which outputs at the end of round $R+2$. From the above claim, it follows that all non-crashed processes simulate synchronous views of the first R rounds. Therefore all non-crashed processes will receive an output from the synchronous protocol $\mathcal{A}$. Moreover, these synchronous views of processes are consistent with a synchronous execution, therefore the set of outputs received by non-crashed processes verifies the predicate $\mathcal{P}$. Hence all the inputs that the processes propose to the Backup protocol verify the predicate $\mathcal{P}$. Since Backup respects validity, it follows that the outputs of Backup will also verify the predicate $\mathcal{P}$, since the task we consider is colorless. This concludes the proof of the output predicate condition. □

We now prove that in a synchronous execution, every process decides by round $R+2$, and no process sends any messages after round $R+2$, ensuring quiescence.

Lemma 7. *In a synchronous execution prefix of $R+2$ rounds, every process decides by the end of round $R+2$, and no process sends any messages after round $R+2$.*

Proof. Since the execution prefix is synchronous, each correct process receives a YES notification from the detector at the end of round $R+2$, and therefore decides. It follows that each correct process starts running the Backup algorithm in *decided* state, keeping its decision. All the backup algorithms we consider, i.e. Paxos [16], ASAP [3], or K4 [2], have the property that a process in decided state sends no additional messages (and in fact takes no additional steps) unless it receives a message from a process that is not in decided state. Since *all* the processes are in decided state, it follows that no process sends any message while running the backup protocol, ensuring quiescence.

We can therefore conclude that the transformation generates a correct indulgent algorithm at the cost of two additional communication rounds.

Theorem 2. *Given a synchronous algorithm $\mathcal{A}$ solving a colorless task T in R communication rounds, and a backup algorithm* Backup *such as [2, 3, 16], the transformation yields an indulgent algorithm that preserves safety in every execution, that requires $R + 2$ rounds to decide in synchronous executions.*

6 A Protocol for Strong Indulgent Renaming

6.1 Protocol Description

In this section, we present an emulation technique that transforms any synchronous renaming protocol into an indulgent renaming protocol. For simplicity, we will assume that the synchronous renaming protocol is the one by Chaudhury et al. [6], which is time-optimal, terminating in $\lceil \log N \rceil + 1$ synchronous rounds. The resulting indulgent protocol will rename in N names using $\lceil \log N \rceil + 3$ rounds of communication if the system is initially synchronous, and will eventually rename into $N + t$ names if the system is asynchronous, by safely reverting to a backup constituted by the asynchronous renaming algorithm by Attiya et al. [4]. Again, the protocol is structured into two phases.

First Phase. During the first $\lceil \log N \rceil + 1$ rounds, processes run the $\mathsf{AD}(2)$ asynchrony detector in parallel with the synchronous renaming algorithm. If the detector returns NO at one of these rounds, then the process stops running the synchronous algorithm, and continues only with the detector. If at the end of round $\lceil \log N \rceil + 1$, the process receives YES from $\mathsf{AD}(2)$, then it also receives a name $name_i$ as the decision value of the synchronous protocol.

Second Phase. At the end of round $\lceil \log N \rceil + 1$, the processes start the asynchronous renaming algorithm of [4]. More precisely, each process builds a vector V with a single entry, which contains the tuple $\langle v_i, name_i, J_i, b_i, r_i \rangle$, where v_i is the processes' initial value. The entry $name_i$ is the proposed name, which is either the name returned by the synchronous renaming algorithm, if the process received YES from the detector, or $\perp$, otherwise. The entry J_i counts the number of times the process proposed a name–it is 1 if the process has received YES from the detector, and 0 otherwise; b_i is the decision bit, which is initially 0. Finally, r_i is the round number when the entry was last updated, which is in this case $\lceil \log N \rceil + 1$. (This last entry in the vector is implied in the original version of the algorithm in [4].)

The processes broadcast their vectors V for the next two rounds, while continuing to run the asynchrony detector in parallel. The contents of the vector V are updated at every round, as follows: if a vector V' containing new entries is received, the process adds all the new entries to its vector; if there are conflicting entries corresponding to the same process, the tie is broken using the round number r_i.

If, at the end of round $\lceil \log N \rceil + 3$ the process receives YES from the detector, then it decides on $name_i$. Regardless of whether it decides or not at the end of this round, each process continues running the asynchronous renaming protocol of [4] as a backup. (Note that, in a synchronous execution, all processes decide by the end of round $\lceil \log N \rceil + 3$, and therefore no more steps are taken as part of the asynchronous renaming protocol.)

6.2 Proof of Correctness

The first step in the proof of correctness of the transformation provides some properties of the asynchronous renaming algorithm of [4]. More precisely, the first Lemma states that the asynchronous renaming algorithm remains correct even though processes propose names initially, that is at the beginning of round $\lceil \log N \rceil + 2$. The proof follows from a simple examination of the protocol and proofs from [4].

Lemma 8. *The asynchronous renaming protocol of [4] ensures termination, name uniqueness, and a name space bound of $N + t$, even if processes propose names at the beginning of the first round of the execution. Also, a process that has decided on a name will not send any messages for the remainder of the execution, unless it receives a message from a process that has not yet decided.*

The previous Lemma ensures that the transformation guarantees weak termination, i.e. that every correct process eventually decides a name. The non-triviality property of the asynchrony detector ensures that every process will decide in $\lceil \log N \rceil + 3$ rounds in any synchronous run. By an identical argument to the one in Lemma 7, we obtain the algorithm is quiescent after $\lceil \log N \rceil + 3$ synchronous rounds, based on the properties of the asynchronous renaming protocol of [4]. Therefore, in the following, we will concentrate on name uniqueness and namespace bounds. We start by proving that the protocol does not generate duplicate names.

Lemma 9 (Uniqueness). *Given any two names n_i, n_j returned by processes in an execution, we have that $n_i \neq n_j$.*

Proof. Assume for the sake of contradiction that there exists a run in which two processes p_i, p_j decide on the same name n_0. First, we consider the case in which both decisions occurred at round $\lceil \log N \rceil + 3$, the first round at with a process can decide using our emulation. Notice that, if a decision is made, the processes necessarily decide on the decision value of the simulated synchronous protocol. (Note that a simple examination of the asynchronous renaming protocol of [4] shows that a process cannot decide after two rounds of communication, unless it had already proposed a value at the beginning of the first round.)

By the global detection property of $\mathsf{AD}(2)$ it then follows that there exists a synchronous execution of the synchronous renaming protocol in which two distinct processes return the same value, contradicting the correctness of the protocol. Similarly, we can show that if both decisions occur *after* round $\lceil \log N \rceil + 3$, we can reduce the correctness of the transformation to the correctness of the asynchronous protocol.

Therefore, the remaining case is that in which one of the decisions occurs at round $\lceil \log N \rceil + 3$, and the other decision occurs at a later round, i.e. it is a decision made by the asynchronous renaming protocol. In this case, let p_i be the process that decides on name n_0 at the end of round $\lceil \log N \rceil + 3$. This implies that process p_i received YES at the end of round $\lceil \log N \rceil + 3$ from $\mathsf{AD}(2)$. Therefore, since p_i sees a synchronous view, there exists a set S of at least $N - t$ processes that received p_i's message reserving name n_0 in round $\lceil \log N \rceil + 2$. It then follows that each non-crashed process receives a message from a process in S in round $\lceil \log N \rceil + 3$. By the structure of the protocol, we obtain that each process has the entry $\langle v_i, n_0, 1, 0, \lceil \log N \rceil + 1 \rangle$ in their V vector at the end of round $\lceil \log N \rceil + 3$. It follows from the structure of the asynchronous protocol of [4] that no process other than p_i will ever decide on the name n_0 at any later round, which concludes the proof of the Lemma. □

We prove that the transformation ensures the following guarantees on the size of the namespace.

Lemma 10 (Namespace Size). *The transformation ensures the following properties: (1) In synchronous executions, the resulting algorithm will rename in a namespace of at most N names. (2) In any execution, the resulting algorithm will rename in a namespace of at most $N + t$ names.*

Proof. For the proof of the first property, notice that, in a synchronous execution, any output combination for the transformation is an output combination for the synchronous renaming protocol.

For the second property, let $\ell \geq 0$ be the number of names decided on at the end of round $\lceil \log N \rceil + 3$ in a run of the protocol. These names are between 1 and N, since the synchronous protocol we simulate [6] solves strong renaming. Lemma 9 guarantees that none of these names is decided on in the rest of the execution. On the other hand, Lemma 8 and the namespace bound of $N + t$ for the asynchronous protocol ensure that the asynchronous protocol decides exclusively on names between 1 and $N + t$, which concludes the proof of the claim. □

We can therefore conclude that the protocol is a correct renaming protocol, giving a tight namespace in synchronous executions, and a namespace of $N + t$ names otherwise.

Theorem 3. *The given protocol solves renaming in $\lceil \log N \rceil + 3$ rounds and N names if the system is synchronous, and in $N + t$ names otherwise.*

7 Conclusions and Future Work

In this paper, we have introduced a general transformation technique from synchronous algorithms to indulgent algorithms, and applied it to obtain indulgent solutions for distributed tasks, such as consensus, set agreement and

renaming. Our results suggest that, even though it is generally hard to design asynchronous algorithms in fault-prone systems, one can obtain efficient algorithms that tolerate asynchronous executions starting from synchronous algorithms.

In terms of future work, we first envision generalizing our technique to generate algorithms that also work in a window of synchrony, and investigating its limitations in terms of time and communication complexity. Another interesting research direction would be to analyze if similar techniques exist in the case of Byzantine failures–in particular, if, starting from a synchronous fault-tolerant algorithm, one can obtain a Byzantine fault-tolerant algorithm, tolerating asynchronous executions.

8 Acknowledgements

The authors would like to thank Hagit Attiya and Nikola Knežević for their feedback on previous drafts of this paper, and the anonymous reviewers for their useful comments.

References

1. Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, LNCS 1320. Springer-Verlag, September 1997.
2. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Of choices, failures and asynchrony: The many faces of set agreement. In *ISAAC 2009*.
3. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. How to solve consensus in the smallest window of synchrony. In *DISC*, pages 32–46, 2008.
4. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
5. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *ACM Symposium on Principles of Distributed Computing*, pages 325–340, August 1991.
6. Soma Chaudhuri, Maurice Herlihy, and Mark R. Tuttle. Wait-free implementations in message-passing systems. *Theor. Comput. Sci.*, 220(1):211–245, 1999.
7. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The disagreement power of an adversary. In *DISC*, pages 8–21, 2009.
8. Partha Dutta and Rachid Guerraoui. The inherent price of indulgence. In *PODC '02: Proceedings of the annual ACM symposium on Principles of distributed computing*, pages 88–97, 2002.
9. Partha Dutta and Rachid Guerraoui. The inherent price of indulgence. *Distributed Computing*, 18(1):85–98, 2005.
10. Partha Dutta, Rachid Guerraoui, and Idit Keidar. The overhead of consensus failure recovery. *Distributed Computing*, 19(5-6):373–386, 2007.
11. Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
12. E. Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *Proceedings of the 17th Symposium on Principles of Distributed Computing*, 1998.
13. Rachid Guerraoui. Indulgent algorithms. In *PODC' 2000*, pages 289–297. ACM, July 2000.
14. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(2):858–923, 1999.
15. Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *PODC*, pages 169–178, 2006.
16. Leslie Lamport. Generalized consensus and Paxos. *Microsoft Research Technical Report MSR-TR-2005-33*, March 2005.
17. Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.