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, 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 synchronization into an algorithm that tolerates asynchronous executions. We introduce a transformation technique from synchronous algorithms to indulgent algorithms [1], 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 a large class of *colorless* 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 feasibility and complexity of distributed tasks has been thoroughly studied both in the synchronous and asynchronous models. To better capture the properties of real-world systems, Dwork, Lynch, and Stockmeyer [2] proposed the *partially synchronous* model, in which the distributed system may alternate between synchronous to asynchronous periods. This line of research inspired the introduction of *indulgent* algorithms [1], 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., [3, 4]). However, designing and proving correctness of such algorithms is usually a difficult task, especially if the algorithm has to provide good performance guarantees.

Contribution. 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 a large class of *colorless* agreement tasks, including consensus and set agreement. We also apply our transformation to the distinct class of *colored* tasks, to obtain the first indulgent renaming algorithm.

Detecting Asynchrony. 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.

The Transformation Technique. 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.

Transforming Agreement Algorithms. We first showcase the technique by transforming algorithms for a large class of agreement tasks, 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 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 it 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 [4, 5] or ASAP [6], 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.

Transforming Renaming Algorithms. We also apply our technique to the renaming problem [7], and obtain the first *indulgent* renaming algorithm. Starting from the synchronous protocol of [8], our protocol renames in a tight namespace of N names and terminates in $(\log N + 3)$ rounds, in synchronous executions. In asynchronous executions, the protocol renames in a namespace of size $N + t$.

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. Due to space limitations, the proofs of some basic results are omitted, and we present detailed sketches for some of the proofs.

2 Model

We consider an eventually synchronous system with N processes $\Pi = \{p_1, p_2, \ldots, p_N\}$, in which $t < N/2$ processes may fail by crashing. Processes communicate via message-passing in rounds, which we model much as in [3, 9, 10]. In particular, time is divided into rounds, which are synchronized. However, the system is asynchronous, i.e. there is no guarantee that a message sent in a round is also delivered in the same round. We do assume that processes receive at least $N - t$ messages in every round, and 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 is delivered in the same round in which it was sent. We denote such a system by $\mathsf{ES}(\mathsf{N}, \mathsf{t})$.

Although indulgent algorithms are designed to work in this asynchronous 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.

In the following, we say that an execution is *synchronous* if every message sent by a correct process in the course of the execution is delivered in the same round in which it was sent. Alternatively, if process p_i receives a message m from process p_j in round $r \geq 2$, then every process received all messages sent by process p_j in all rounds $r' < r$. The *view* 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 the *view* of process p is *synchronous* at round r if there exists an r-round synchronous execution which is indistinguishable from p's view at round r.

3 Related Work

Starting with seminal work by Dwork, Lynch and Stockmeyer [2], a variety of different models have been introduced to express relaxations of the standard asynchronous model of computation. These include failure detectors [11], round-by-round fault detectors (RRFD) [12], and, more recently, indulgent algorithms [1].

In [3, 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.

Recent work studied the complexity of agreement problems, such as consensus [6] and k-set agreement [10], if the system becomes synchronous after an unknown stabilization time GST. In [6], 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 [10], 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, the transformation in the current paper does not immediately yield algorithms that would work in a window of synchrony. From the point of view of the technique, references [6, 10] 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 [13] considered a setting in which failures stop after GST, in which case 3 rounds of communication are necessary and sufficient. Leader-based, Paxos-like algorithms, e.g. [4, 5], form another class of algorithms that tolerate asynchrony, and can also be seen as indulgent algorithms.

A precise definition of colorless tasks is given in [14]. Note that, in this paper, we augment their definition to include the standard *validity* property (see Section 5).

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 the end of 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, in every round r, returns either* YES *or* NO*, at each process. The detector ensures the following properties.*

- *(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.*
- *(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 $r - d$.*
- *(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 $(r - R)$-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 main idea behind the detector, implemented in the process procedure, is that processes maintain a detailed view of the state of the system by aggregating all messages received in every round. For each round, each process maintains an $Active$ set of processes, i.e. processes that sent at least one 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 8–9). 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 10–12). In the next round, each process sends its updated view of the system together with a $synch$ flag, which was set to true, if asynchrony was detected.

```
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 )
5      msgSet_i ← receive()
6      (synch_i, msg_i) ← process(msgSet_i, R_c)
7      if synch_i = true then output YES
8      else output NO

1  procedure process( msgSet_i, r )_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      for every msg_j ∈ msgSet_i do
7        for round r from 1 to R_c do
8          Active_i[r] ← msg_j.Active_j[r] ∪ Active_i[r]
9          Failed_i[r] ← msg_j.Failed_j[r] ∪ Failed_i[r]
10     for round r from 1 to R_c − 1 do
11       for round k from r + 1 to R_c do
12         if (Active_i[k] ∩ Failed_i[r] ≠ ∅) then synch_i ← false
13   if synch_i = true then
14     msg_i ← (synch_i, (Active_i[r])_{r∈[1,R_c]}, (Failed_i[r])_{r∈[1,R_c]})
15   else msg_i ← (synch_i, ⊥, ⊥)
16   return (synch_i, msg_i)
```

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

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. First, 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 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 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 begin by proving that if two processes receive YES from the asynchronous detector in round $r + 2$, then they must have received eachother's round $r + 1$ messages, either directly, or through a relay. 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 $Active[r + 2]$ 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 $p \in Active_q^{r+2}[r + 1]$ and $q \in Active_p^{r+2}[r + 1]$.*

Proof. We prove that $p \in Active_q^{r+2}[r + 1]$–the proof of the second statement is symmetric. Assume, for the sake of contradiction, that $p \notin Active_q^{r+2}[r + 1]$. Then, by lines 8–9 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.

Lemma 2. *Let p and q be two processes in P. Then, for all rounds $k < l \leq r$, $Active_p^r[l] \cap Failed_q^r[k] = \emptyset$, and $Active_p^r[l] \cap Failed_q^r[k] = \emptyset$, 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 processor s such that $s \in Active_p^r[l] \cap Failed_q^r[k]$. Lemma 1 ensures that p and q communicate in round $r + 1$, therefore it follows 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 in* ES(N, t)*, and P be a set of processes in Π such that, at the end of round R, the following two properties are satisfied:*

1. *For any p and q in P, and any round $r \in \{1, 2, \ldots, R - 1\}$, $Active_p^R[r + 1] \cap Failed_q^R[r] = \emptyset$.*
2. *$|\bigcap_{p \in P} Active_p^R[R]| \geq N - t$.*

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

Finally, we prove that if a set of processes P receive YES from $AD(2)$ 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 $R > 0$, i.e. that $AD(2)$ is indeed a 2-round asynchrony detector. The proof follows from the previous results.

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

5 Generating Indulgent Algorithms for Colorless Tasks

5.1 Task Definition

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 input or output 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 or k-agreement. 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.

5.2 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 communication rounds. 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 an 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, e.g. Paxos [5], or ASAP [6].

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 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[4].

On the other hand, if the process receives NO from $AD(2)$ in round $R+2$, i.e. asynchrony was detected, then the process will run the second phase of the transformation. More precisely, in phase two, the process will run a backup agreement protocol that tolerates periods of asynchrony (for example, the K4 protocol [10], if the task is k-set agreement). 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 $Supp$ (the *support* set) be the set of processes that received YES from 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 from $Supp$ and adopts its state at the end of round $R-1$. Then, in round R, it simulates receiving the messages in $\bigcap_{j \in Supp} msgSet_j^{R+1}[R]$, where we maintain the notation used in Section 4. 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.

5.3 Proof of Correctness

We now prove that the resulting protocol verifies the task specification. The proofs of termination, validity, and the colorless property follow 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 $\mathcal{P}$.

Theorem 1 (Output Predicate). *The indulgent transformation protocol satisfies the output predicate $\mathcal{P}$ associated to the task T.*

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 AD(2), there exists a synchronous execution of $\mathcal{A}$ in which the same outputs are decided, which break the predicate $\mathcal{P}$, contradiction. If all decisions occur after round $R+2$, first notice that, by the validity and colorless properties, the inputs processes propose to the

[4] 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.

Backup protocol are always valid inputs for the task. It follows that, since all decisions are output by Backup, 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. If a process decides at the end of round $R+2$, then (i) all correct processes will have a non-empty support set $Supp$ and (ii) there exists an R-round synchronous execution consistent with the views that all correct processes adopt at the end of round $R+2$.

Proof (Sketch). 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 $R+1$. 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 $R+1$; therefore, all non-crashed processes that get NO from $\mathsf{AD}(2)$ in round $R+2$ will execute case 2, which ensures the first claim.

Let $Q = \{q_1, \ldots, q_\ell\}$ be 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. To prove that the intersection of their simulated views in round R contains at least $(N-t)$ messages, notice that the processes from which process d receives messages in round $R+2$ are necessarily in this intersection, since otherwise process d would receive NO in round $R+2$.

To prove the first condition of Lemma 3, note that process d's view of round R, i.e. the set $msgSet_d^{R+2}[R]$, contains all messages simulated as received in round R by the processes that receive NO in round $R+2$. Since $N-t > t$, every process that receives NO in round $R+2$ from the detector also receives a message supporting d's decision in round $R+2$; process d receives the same message and does not notice any asynchrony.

Therefore, 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, recall that there exists at least 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 $\mathcal{P}$.

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 Herlihy et al. [8], 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. [7]. Again, the protocol is structured into two phases.

First Phase. During the first $\lceil \log N \rceil + 1$ rounds, processes run the $AD(2)$ asynchrony detector in parallel with the synchronous renaming algorithm. Note that the protocol's messages are included in the detector's messages. 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 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 [7]. 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[5] when the entry was last updated, which is in this case $\lceil \log n \rceil + 1$.

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 $name_i$. Otherwise, it continues runnning the AttiyaRenaming algorithm until decision is possible.

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 [7]. 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 an examination of the protocol and proofs from [7].

Lemma 5. *The asynchronous renaming protocol of [7] ensures termination, name uniqueness, and a name space bound of $N + t$, even if processes propose names at the beginning of the first round.*

The previous Lemma ensures that the transformation guarantees termination, i.e. that every correct process eventually returns a name. The non-triviality property of the

[5] This entry in the vector is implied in the original version of the algorithm [7].

asynchrony detector ensures that the resulting algorithm will terminate in $\lceil \log N \rceil + 3$ rounds in any synchronous run. In the following, we will concentrate on the *uniqueness* of the names and on the bounds on the resulting namespace. We start by proving that the protocol does not generate duplicate names.

Lemma 6 (Uniqueness). *Given any two names n_i, n_j returned by processes in an execution, we have that $n_i \neq n_j$.*

Proof (Sketch). 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[6]. 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 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 the set 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 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.

Finally, we prove that the transformation ensures the following guarantees on the size of the namespace.

Lemma 7 (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 (Sketch). 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 clearly between 1 and N. Lemma 6 guarantees that none of these names is decided on in the rest of the execution. On the other hand, Lemma 5 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.

[6] A simple analysis of the asynchronous renaming protocol 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.

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 a large class of distributed tasks, including 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 Prof. Hagit Attiya and Nikola Knežević for their help on previous drafts of this paper, and the anonymous reviewers for their useful feedback.

References

1. R. Guerraoui, "Indulgent algorithms," in *PODC' 2000*, pp. 289–297, ACM, July 2000.
2. C. Dwork, N. A. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, pp. 288–323, Apr. 1988.
3. P. Dutta and R. Guerraoui, "The inherent price of indulgence.," in *PODC '02: Proceedings of the annual ACM symposium on Principles of distributed computing*, pp. 88–97, 2002.
4. L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.
5. L. Lamport, "Generalized consensus and paxos," *Microsoft Research Technical Report MSR-TR-2005-33*, March 2005.
6. D. Alistarh, S. Gilbert, R. Guerraoui, and C. Travers, "How to solve consensus in the smallest window of synchrony," in *DISC*, pp. 32–46, 2008.
7. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, "Renaming in an asynchronous environment," *Journal of the ACM*, vol. 37, no. 3, pp. 524–548, 1990.
8. S. Chaudhuri, M. Herlihy, and M. R. Tuttle, "Wait-free implementations in message-passing systems," *Theor. Comput. Sci.*, vol. 220, no. 1, pp. 211–245, 1999.
9. P. Dutta and R. Guerraoui, "The inherent price of indulgence," *Distributed Computing*, vol. 18, no. 1, pp. 85–98, 2005.
10. D. Alistarh, S. Gilbert, R. Guerraoui, and C. Travers, "Of choices, failures and asynchrony: The many faces of set agreement," in *ISAAC 2009*.
11. T. D. Chandra and S. Toueg, "Unreliable failure detectors for asynchronous systems (preliminary version)," in *ACM Symposium on Principles of Distributed Computing*, pp. 325–340, Aug. 1991.
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. P. Dutta, R. Guerraoui, and I. Keidar, "The overhead of consensus failure recovery," *Distributed Computing*, vol. 19, no. 5-6, pp. 373–386, 2007.
14. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann, "The disagreement power of an adversary," in *DISC*, pp. 8–21, 2009.