You Only Live Multiple Times: A Blackbox Solution for Reusing Crash-Stop Algorithms In Realistic Crash-Recovery Settings*

David Kozhaya
ABB Corporate Research, Switzerland
david.kozhaya@ch.abb.com

Ognjen Marić
Digital Asset, Switzerland
ogi.yolmt@mynosefroze.com

Yvonne-Anne Pignolet
ABB Corporate Research, Switzerland
yvonne-anne.pignolet@ch.abb.com

Abstract

Distributed agreement-based algorithms are often specified in a crash-stop asynchronous model augmented by Chandra and Toueg's unreliable failure detectors. In such models, correct nodes stay up forever, incorrect nodes eventually crash and remain down forever, and failure detectors behave correctly forever eventually, However, in reality, nodes as well as communication links both crash and recover without deterministic guarantees to remain in some state forever.

In this paper, we capture this realistic temporary and probabilitic behaviour in a simple new system model. Moreover, we identify a large algorithm class for which we devise a property-preserving transformation. Using this transformation, many algorithms written for the asynchronous crash-stop model run correctly and unchanged in real systems.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Crash recovery, consensus, asynchrony

1 Introduction

Distributed systems comprise multiple software and hardware components that are bound to eventually fail [10]. Such failures can cause service malfunction or unavailability, incurring significant costs to mission-critical systems, e.g., automation systems and on-line transactions. The failures' impact can be minimized by protocols that let systems agree on actions despite failures. As a consequence, many variants of the agreement or the consensus problem [29] under different assumptions have been studied. Of particular importance are synchrony and failure model assumptions, as they determine the problem's complexity.

In the simplest failure model, often called the *crash-stop* model, a process fails by stopping to execute its protocol and never recovers. Combining this failure model with an asynchronous system, i.e., a system without bounds on execution delays or message latency, makes it impossible to distinguish a crashed from a very slow process. This renders consensus-like problems unsolvable deterministically [16], already in this very simple failure model. To circumvent this impossibility, previous works have investigated ways to relax the underlying asynchrony assumption either explicitly, e.g., by using partial synchrony [12], or implicitly, by defining oracles that encapsulate time, e.g., failure detectors [8]. The result

* Full version available at `https://arxiv.org/abs/1811.05007`

is a large and rich body of literature that builds on top of the former and latter techniques to solve consensus-like problems in the presence of crash-stop failures. Typically, the respective proofs rely on assumptions of the "eventually forever" form: the correct nodes stay up forever, incorrect nodes eventually crash and remain down forever, and failure detectors produce wrong output in the beginning, but provide correct results forever eventually.

However, such "eventually forever" assumptions are not met by real distributed systems. In reality, processes may crash but their processors reboot, and the recovered process rejoins the computation. Communication might also fail at any point in time, but get restored later. Hence, the failure and recovery modes of processes as well as communication links are in reality probabilistic and temporary [13, 15, 31], especially in systems incorporating many unreliable off-the-shelf low-cost devices and communication technologies. This led to the development of *crash-recovery* models, where processes repeatedly leave and join the computation unannounced. This requires new failure detector definitions and new consensus algorithms built on top of these failure detectors [1, 24, 11, 21] as well as completely new solutions (without failure detectors) that consider different classes of failures, namely classified according to how many times a process can crash and recover [25]. However, such solutions eliminate the "eventually forever" assumptions only on the processes' level and not for the communication and failure detectors. Moreover, these works derive new algorithms tailored to crash-recovery settings. This leaves unanswered the question: can the plethora of existing crash-stop algorithms be reused (ideally, unchanged) in crash-recovery settings?

To this end, this paper investigates how to re-use consensus algorithms defined for the crash-stop model with reliable links and failure detectors in a more realistic crash-recovery model, where processes and links can crash and recover probabilistically and for an unbounded number of times. Our models allow *unstable* nodes, i.e., nodes that fail and recover infinitely often. These are often excluded or limited in number in other models. In contrast, we explicitly allow unstable behavior of any number of processes and links, by modeling communication problems and crash-recovery behaviors as probabilistic and temporary, rather than deterministic and perpetual. Our system model, similar to existing models that rely on probabilistic factors, e.g., coin flips, comes with the trade-off of solving consensus (namely the termination property), with probability 1, rather than deterministically.

However, unlike existing solutions that incorporate probabilistic behavior, our approach does not aim at inventing new consensus algorithms but rather focuses on using existing deterministic ones to solve consensus with probability 1. Our approach is modular: we build a wrapper that interacts with a crash-stop algorithm as a black box, exchanges messages with other wrappers and transforms these messages into messages that the crash-stop algorithm understands. We then formally define classes of algorithms and safety properties for which we prove that our wrapper constructs a system that preserves these properties. Additionally, we show that termination with probability 1 is guaranteed for wrapped algorithms of this class. Moreover, this class is wide and includes the celebrated Chandra-Toueg algorithm [8] as well as the instantiation of the indulgent framework with failure detectors from [20] . Our work allows such algorithms to be ported unchanged to our crash-recovery model. Hence applications built on top of such algorithms can run in real systems with crash-recovery behavior by simply using our wrapper.

Contributions: To summarize, our main contributions are:

- New system models that capture probabilistic and temporary failures and recoveries of processes and communication links in real distributed systems (described in Section 3)
- A wrapper framework that allows a wide class of crash-stop consensus algorithms to be used unchanged in our more realistic models (described in Section 4)

- Formal properties describing which crash-stop consensus algorithms benefit from our framework and hence can be reused to solve consensus in crash-recovery settings (described in Sections 5 and 6)

In addition to the sections presenting our contributions, we discuss related work in Section 2 and conclude the paper in Section 7. Due to space limitations, we defer most proofs and some formalization details to the full version, available at `https://arxiv.org/abs/1811.05007`.

2 Related Work

Several works addressed the impossibility of asynchronous consensus. One direction exploits the concept of partial synchrony [12], in which an asynchronous system becomes synchronous after some unknown global stabilization time (GST) for a bounded number of rounds. For the same model, ASAP [3] is a consensus algorithm where every process decides no later than round $GST+f+2$ (optimal). Another direction augments asynchronous systems with failure detector oracles, and builds asynchronous consensus algorithms on top [8]. These detectors typically behave erratically at first, but eventually start behaving correctly forever. Like with partial synchrony, the intuitive expectation is that failure detectors must only behave correctly for "sufficiently long" instead of forever [8]; however, quantifying "sufficiently long" is impossible in a purely asynchronous model [9]. Both lines of work initially investigated crash-stop failures of processes. In real systems processes as well as network links crash and recover multiple times and sometimes even indefinitely. This gave rise to a large body of literature that studied how to adapt the two lines of work to crash-recovery behavior of processes and links. We next survey some of this literature.

Failure detectors and consensus algorithms for crash recovery: Dolev et al. [11] consider an asynchronous environment where communication links first lose messages arbitrarily, but eventually communication stabilizes such that a majority of processes forms a forever strongly connected component. Processes belonging to such a strongly-connected component are termed correct, and the others faulty. Process state is always (fully) persisted in stable storage. The authors propose a failure detector that allows the correct processes to reach a consensus decision and show that the rotating coordinator algorithm [8] works unchanged in their setting, as long as all messages are constantly retransmitted. This relies on piggybacking all previous messages onto the last message, and regularly retransmitting the last message. As this yields very large messages, they also propose a modification of [8] for which no piggybacking is necessary. While our results also rely on strongly connected components, we do not require their existence to be deterministic nor perpetual. We also do not require piggybacking in order for algorithms like [8] to be used unchanged.

Oliveira et al. [28] consider a crash-recovery setting with correct processes that may crash only finitely many times (and thus eventually stay up forever) and faulty processes that permanently crash or crash infinitely often. As in [8], the authors note that correct processes only need to stay up for long enough periods in practice (rather than forever), but this cannot be expressed in the asynchronous model. The authors take the consensus algorithm of [30] which uses stubborn links and transform it to work in the crash-recovery setting by logging every step into stable storage and adding a fast-forward mechanism for skipping rounds. Hurfin et al. [22] describe an algorithm using the $\diamond\mathcal{S}$ detector in the crash-recovery case. The notions of correct/faulty processes and of failure detectors are the same as in Oliviera et al [28]. Their algorithm is however more efficient when using stable storage compared to [28]: there is only one write per round (of multiple data), and the required network buffer capacity for each channel (connecting a pair of processes) is one. Compared

to [28] and [22] our system does not regard processes that crash and recover infinitely often as faulty and hence we allow such "unstable" processes to implement consensus.

Aguilera et al. [1] consider a crash-recovery system with lossy links. They show that previously proposed failure detectors for the crash-recovery setting have anomalous behaviors even in synchronous systems when considering unstable processes, i.e., processes that crash and recover infinitely often. The authors propose new failure detectors to mitigate this drawback. They also determine the necessary conditions regarding stable storage that allow consensus to be solved in the crash-recovery model, and provide two efficient consensus algorithms: one with, and one without using stable storage. Unlike [1], we do not exclude unstable processes from implementing consensus, thus our model tolerates a wider variety of node behavior. Furthermore, our wrapper requires no modifications to the existing crash-stop consensus algorithms, as it treats them as black-boxes.

Modular Crash-Recovery Approaches: Similar to [1], Freiling et al. [18] investigate the solvability of consensus in the crash-recovery model under varying assumptions, regarding the number of unstable and correct processes and what is persisted in stable storage. They reuse existing algorithms from the crash-stop model in a modular way (without changing them) or semi-modular way, with some modifications to the algorithm (as in the case of [8]). Similar to our work, they provide algorithms to emulate a crash-stop system on top of a crash-recovery system. Our work, however, always reuses algorithms in a fully modular way, and we define a wide class of algorithms for which such reuse is possible. Furthermore, as we model message losses, processes crashes, and process recoveries probabilistically, our results also apply if processes are unstable, i.e., crash and recover infinitely often.

Randomized Consensus Algorithms: Besides the literature that studied deterministic consensus algorithms, existing works have also explored randomized algorithms to solve "consensus with probability 1". These include, for example, techniques based on using random *coin-flips* [2, 5, 17] or *probabilistic schedulers* [7]. In systems with dynamic communication failures, multiple randomized algorithms [27, 26] addressed the *k-consensus* problem, which requires only k processes to eventually decide. Moniz et al. [27] considered a system with correct processes and a bound on the number of faulty transmission. In a wireless setting, where multiple processes share a communication channel, Moniz et al. [26] devise an algorithm tolerating up to f Byzantine processes and requires a bound on the number of omission faults affecting correct processes. In comparison, our work in this paper does not use randomization in the algorithm itself: we focus on using existing deterministic algorithms to solve consensus (with probability 1) in networks with probabilistic failure and recovery patterns of processes and links.

3 System Models

We start by defining the notation we use, and then define general concepts common to all of our models. Then, we define each of our models in turn.

Notation: Given a set S, we define $S_\perp$ to be the set $S \cup \{\perp\}$, where $\perp$ is a distinguished element not present in S. The set of finite sequences over a set S is denoted by S^*. We also call sequences *words*, when customary. Given a non-empty sequence, *head* defines its first element, *tail* the remainder of the sequence, and, if the sequence is finite, *last* its last element. Given two sequences u and v, where u is finite, $u{\cdot}v$ denotes their concatenation. For a word u, $|u|$ denotes the length of u. Letting $u(i)$ be the i-th letter of u, we say that u is a *subword* of v if there exists a strictly monotone function $f : \mathbb{N} \to \mathbb{N}$ such that $u(i) = v(f(i))$, for $1 \leq i \leq |u|$ if u is finite, and for all $i \in \{1, 2, \ldots\}$ if u is infinite. Analogously, v is a

superword of u.

We denote the space of partial functions (maps) between sets A and B by $A \rightharpoonup B$. Note that $(A \to B) \subseteq (A \rightharpoonup B)$.

Common concepts: We consider a fixed finite set of processes $\Pi = \{1 \dots N\}$, and a fixed countable set of values, denoted $\mathcal{V}$. For each algorithm there is an algorithm-specific countable set of local states Σ_p, for each process $p \in \Pi$. For simplicity, we restrict ourselves to algorithms where $\Sigma_p = \Sigma_q,\ \forall p, q \in \Pi$. Note that this does not exclude algorithms that take decisions based on identifiers. We define the global state space $\Sigma = \prod_{p \in \Pi} \Sigma_p$. Given a $s \in \Sigma$, we define $s_p \in \Sigma_p$ as the projection of s to its p-th component.

A *property* over an alphabet A is a set of infinite words over A. We use standard definitions of liveness and safety properties [4]. A property P is a *safety property* if, for every infinite word $w \notin P$, there exists a finite prefix u of w such that the concatenation $u \cdot v \notin P$ for all infinite words v. Intuitively, the prefix u is "bad" and not recoverable from. A property P is a *liveness property* if for any finite word u there exists an infinite word v such that $u \cdot v \in P$. Intuitively, "good" things can always happen later.

In this paper, we are interested in preserving properties over the alphabet Σ between the crash-stop and crash-recovery versions of an algorithm. In particular, we assume that the local states Σ_p are records, with two distinguished fields: *inp* of type $\mathcal{V}$ and *dec* of type $\mathcal{V}_\perp$. Intuitively, a *dec* value of $\perp$ indicates that the process has not decided yet. For an infinite word w over the alphabet Σ, let $w(i, p)$ denote the local state of the process p at the i-th letter of the word. Let us state the standard safety properties of consensus in our notation.

Validity. Decided values must come from the set of input values. Formally, validity describes the set of words w such that $\forall p,\ i,\ v.\ w(i,p).dec = v \wedge v \neq \perp \implies \exists q.\ w(1,q).inp = v$

Integrity. Processes do not change their decisions. Formally, integrity describes the set of words w such that $\forall p,\ i.\ v.\ w(i,p).dec = v \wedge v \neq \perp \implies \forall i' > i.\ w(i',p).dec = v$

(Uniform) Agreement. No two processes ever make different non-$\perp$ decisions. Formally, $\forall p,\ q,\ i,\ j.\ w(i,p).dec \neq \perp \neq w(j,q).dec \implies w(i,p).dec = w(j,q).dec$

To simplify our preservation results for safety properties, our models store information about process failures separately from Σ. As a consequence, the standard crash-stop termination property cannot be expressed as a property over Σ: it is conditioned on a process not failing. However, we do not directly use the crash-stop notion of termination and we omit this definition here. Instead, we will prove the following property for the algorithms in our probabilistic crash-recovery model:

Probabilistic crash-recovery termination. With probability 1, all processes eventually decide.

3.1 The crash-stop model

Our definition of the crash-stop model is standard and closely follows [8]. We assume an asynchronous environment, with processes taking steps in an interleaved fashion. Processes communicate using reliable links, and can query failure detectors.

Failure detectors: A *failure pattern fp* is an infinite word over the alphabet 2^Π. Intuitively, each letter is the set of failed processes in a transition step of a run of a transition system. A *failure detector* with range $\mathcal{R}$ is a function from failure patterns to properties over the

alphabet $\mathcal{R}$.[1] A failure detector D is *unreliable* if $D(fp)$ is a liveness property for all fp. Intuitively, a detector constrains how the failure detector outputs (the $\mathcal{R}$ values) must depend on the failure pattern of a run, and unreliable detectors can produce arbitrary outputs in the beginning. We write $FD(\mathcal{R})$ for the set of all detectors with range $\mathcal{R}$.

Algorithms and algorithm steps: The type of *crash-stop steps* over a message space $\mathcal{M}$ and a failure detector range $\mathcal{R}$, written $CSS(\mathcal{M}, \mathcal{R})$ is defined as a pair of functions of types:

$$next : \Sigma_p \times (\Pi \times \mathcal{M})_\perp \times \mathcal{R} \to \Sigma_p, \quad send : \Sigma_p \to (\Pi \rightharpoonup \mathcal{M}).$$

Intuitively, given zero or one messages received from some other processes and an output of the failure detector, a step maps the current process state to a new state, and maps the new state to a set of messages to be sent, with zero or one messages sent to each process.

A *crash-stop algorithm* $\mathcal{A}$ over Σ, $\mathcal{M}$ and $\mathcal{R}$ is a tuple $(I, step, D, N_f)$ where: (1) $I \subseteq \Sigma$ is the *finite* set of initial states, (2) $step \in CSS(\mathcal{M}, \mathcal{R})$ is the step function, (3) $D \in FD(\mathcal{R})$ is a failure detector, and (4) $N_f < N$ is the resilience condition, i.e., the number of failures tolerated by the algorithm (recall that we consider a fixed N). We refer to the components of an algorithm $\mathcal{A}$ by $\mathcal{A}.I$, $\mathcal{A}.step$, $\mathcal{A}.D$ and $\mathcal{A}.N_f$.

Configurations: As noted earlier, we focus on preserving properties over Σ between crash-stop and crash-recovery models. However, Σ contains insufficient information to model the algorithm's crash-stop executions (runs). In particular, to account for (1) asynchronous message delivery and (2) process failures, we must extend states to *configurations*. A *crash-stop configuration* is a triple (s, M, F) where: $s \in \Sigma$ is the (global) state, $M \subseteq \Pi \times \Pi \times \mathcal{M}$ is the set of *in-flight messages*, where $(p, q, m) \in M$ represents a message m that was sent to p by q, and $F \subseteq \Pi$ is the set of failed processes. As with algorithms, we refer to the components of a configuration c by $c.s$, $c.M$ and $c.F$.

Step labels and transitions: While the algorithm steps are deterministic, the asynchronous transition system is not: any (non-failed) process can take a step at any point in time, with different possible received messages, and different failure detector outputs. Accessing this non-determinism information is useful in proofs, so we extract it as follows. A *crash-stop step label* is a quadruple $(p, rmsg, fails, fdo)$, where: (1) $p \in \Pi$ is the process taking the step, (2) $rmsg \in (\Pi \times \mathcal{M})_\perp$ is the message p receives in the step ($\perp$ modeling a missing message), (3) $fails \subseteq \Pi$ is the set of processes failed at the end of the step, and (4) $fdo \in \mathcal{R}$ is p's output of the failure detector. A *crash-stop step* of the algorithm $\mathcal{A}$ is a triple (c, l, c'), where c and c' are configurations and l is a label. Crash-stop steps must satisfy the following properties: (i) p is not failed at the start of the step. (ii) p takes a step according to the label and $\mathcal{A}$'s rules, and the other processes do not move. (iii) If a message was received, then it was in flight; the received message is removed from the set of in-flight messages, while the produced messages are added. (iv) Failed processes do not recover, i.e., $c.F \subseteq c'.F = fails$.

Algorithm runs: A finite (respectively infinite) *crash-stop run* of $\mathcal{A}$ is a finite (infinite) alternating sequence c_0, l_0, $c_1 \ldots$ of configurations and labels, that ends in a configuration if finite, such that the initial state is allowed by the algorithm, (c_i, l_i, c_{i+1}) are valid steps, and the resilience condition is satisfied. Furthermore, the output of the failure detector must satisfy the condition of the failure detector. Such a run has *reliable links* if all in-flight messages eventually get delivered, unless the sender or the receiver is faulty. The *crash-stop system* of the algorithm $\mathcal{A}$ is the sequence of all crash-stop runs of $\mathcal{A}$. The *crash-stop system with reliable links* of the algorithm $\mathcal{A}$ is the set of all crash-stop runs with reliable links.

[1] This definition does not distinguish which process received the output, which is sufficient for $\diamond\mathcal{S}$. The definition can be easily extended to other failure detectors like $\diamond\mathcal{W}$.

As mentioned before, we are interested in properties that are sequences of global states. In this sense, runs contain too much information (e.g., in-flight messages). Thus, given a run ρ, we define its *state trace* $tr_s(\rho)$, obtained by removing the labels and projecting configurations onto just the states. We introduce a notion of a *state property*: an infinite sequence of (global) states. The crash-stop system (with or without reliable links) satisfies a state property P if for every run ρ of the system, $tr_s(\rho) \in P$. We later show that our crash-recovery wrappers for crash-stop protocols preserve important state properties of crash-stop algorithms. Lastly, we note down a simple property of crash-stop runs.

▶ **Lemma 1** (Reliable links irrelevant for prefixes)**.** *Let ρ be a finite crash-stop run of $\mathcal{A}$. Then, ρ can be extended to an infinite crash-stop run of $\mathcal{A}$ that has reliable links (intuitively, by eventually delivering all in-flight messages, M).*

Summary of time and failure assumptions

Time. Processes are asynchronous and have no notion of time. Links are asynchronous.
Failures. Processes can fail by halting forever, while links interconnecting them do not fail.

3.2 The lossy synchronous crash-recovery model

We next define our first crash-recovery model. Formally, this a lossy synchronous crash-recovery model, with non-deterministic, but not probabilistic losses, crashes, and recoveries. We use it to prove the preservation of safety properties without taking probabilities into account, since they are not used in such arguments. In this model, we will not distinguish between volatile and persistent memory of a process. Instead, we assume that all memory is persistent. This can be emulated in practice by persisting all volatile memory before taking any actions with side-effects (such as sending network messages). Finally, while the model is formally synchronous, in that all processes take steps simultaneously, it also captures processing delays, as a slow process behaves like a process that crashes and later recovers.

Algorithms and algorithm steps: A *crash-recovery step* over a message space $\mathcal{M}$, written $CRS(\mathcal{M})$ is defined as the pair of functions *next* and *send*, with types:

$next : \Sigma_p \times (\Pi \rightharpoonup \mathcal{M}) \to \Sigma_p \quad send : \Sigma_p \to (\Pi \to \mathcal{M})$.

In other words, a step determines the new state based on the current state and the map of received messages. Given the new state, a process sends a message to every other processes (including itself). Compared to the asynchronous setting (Section 3.1), in this model:

1. A process can receive multiple messages simultaneously (rather than receiving at most one message in a step).
2. Every process sends a message to every other process at each step. The wrapped algorithms in later sections satisfy this by sending heartbeat messages, if there is nothing else to exchange. Delivery of sent messages is not guaranteed in this model.
3. No failure detector oracle is specified. The synchrony assumption of this model inherently provides spurious failure detection: each process can suspect all peers it did not hear from in the last message exchange. This is in fact exactly what we will use to provide failure detector outputs to the "wrapped" crash-stop algorithms run in this setting[2].

A *crash-recovery algorithm* $\mathcal{A}^R$ over Σ, $\mathcal{M}$ is a pair $(I, step)$ where: $I \subseteq \Sigma$ is a finite set of initial states, and $step \in CRS(\mathcal{M})$ is the step function.

Configurations: As in the crash-stop case, we require more than just the global states to model algorithm executions; we hence introduce configurations. These, however, differ from

[2] Similar to Gafni's round-by-round fault detectors [19]; in our case, the detectors are "step-by-step"

those for the crash-stop setting. As communication is synchronous, we need not store the in-flight messages; they are either delivered by the end of a step, or they are gone. Furthermore, as processes take steps synchronously, we can introduce a global step number. [3]

A *crash-recovery configuration* is a tuple (n, s, F) where $n \in \mathbb{N}$ is the step number, $s \in \Sigma$ is the (global) state, $F \subseteq \Pi$ is the set of failed processes. We denote the set of all crash-recovery configurations by $\mathcal{C}^R$. Note that this set is countable. This will allow us to impose a Markov chain structure on the system in the later model.

Step labels and transitions: As in the crash-stop setting, we use labels to capture all sources of non-determinism in a step. We will use these labels to assign probabilities to different state transitions in the probabilistic model of the next section.

A *crash-recovery step label* is a pair $(rmsgs, fails)$, where:

- $rmsgs : \Pi \to (\Pi \rightharpoonup \mathcal{M})$ denotes the message received in the step; $rmsgs(p)(q)$ is the message received by p on the channel from q to p. As we assume that q always attempts to send a message to p, if $rmsgs(p)(q)$ is undefined ($rmsgs(p)$ is a partial function), then either the message on this channel was lost in the step, or the sender q has failed.
- $fails \subseteq \Pi$ is the set of processes that are failed at the end of the step.

The *crash-recovery steps* (or transitions) of $\mathcal{A}$, written $Tr(\mathcal{A})$, is the set of all triples (c, l, c'), where (i) only processes that are up handle their messages (ii) messages from failed senders are not received (iii) failed processes $fails = c'.F$.

Algorithm runs: A finite, resp. infinite *crash-recovery run* of $\mathcal{A}$ is a finite, resp. infinite alternating sequence c_0, l_0, $c_1 \ldots$ of (crash-recovery) configurations and labels, ending in a configuration if finite, such that: $c_0.s \in \mathcal{A}.I$, i.e., the initial state is allowed by the algorithm, and $(c_i, l_i, c_{i+1}) \in Tr(\mathcal{A})$ for all i, that is, each step is a valid crash-recovery transition of $\mathcal{A}$. The *crash-recovery system* of algorithm $\mathcal{A}$ is the sequence of all crash-recovery runs of $\mathcal{A}$.

Summary of time and failure assumptions

Time. Processes are synchronous and operate in a time-triggered fashion. Links are synchronous (all delivered messages respect a timing upper bound on delivery).

Failures. Processes can fail and recover infinitely often. In every time-step, a link can be either crashed or correct. A crashed link drop the messages (if any) sent over it.

3.3 The probabilistic crash-recovery model

We now extend the lossy synchronous crash-recovery model to a probabilistic model, where both the successful delivery of messages and failures follow a distribution that can vary with time. A *probabilistic network* $\mathcal{N}$ is a function of type $\Pi \times \Pi \times \mathbb{N} \to [0, 1]$, such that $\exists \epsilon_{\mathcal{N}} > 0.\ \forall p, q, t.\ \mathcal{N}(q, p, t) > \epsilon_{\mathcal{N}}$ Intuitively, $\mathcal{N}(q, p, t)$ is the delivery probability for a message sent from p to q at time (step number) t. A *probabilistic failure pattern* $F^{\mathcal{P}}$ is a function $\Pi \times \mathbb{N} \to [0, 1]$, such that $\exists \epsilon_F > 0.\ \forall p, t.\ \epsilon_F < F^{\mathcal{P}}(p, t) < 1 - \epsilon_F$. Intuitively, $F^{\mathcal{P}}(p, t)$ gives the probability of p being up at time t.[4]

Given a crash-recovery algorithm $\mathcal{A}^R$, a probabilistic network $\mathcal{N}$ and failure pattern $F^{\mathcal{P}}$, a *probabilistic crash-recovery system* $\mathcal{S}^R(\mathcal{A}^R, \mathcal{N}, F^{\mathcal{P}})$ is the Markov chain [6] with:

- The set of states $\mathcal{C}^R$, i.e., the crash-recovery configuration set.
- The transition probabilities $P(c)(c')$, defined as follows. Intuitively, a transition from c to c' is only possible if it is possible in the lossy synchronous crash-recovery model. The

[3] We use global step numbers later in the probabilistic model, to assign failure probabilities for processes and links (e.g., a probability $p_{ij}(t)$ of a message from i to j getting through, if sent in the t-th step).

[4] Considering infinite time, the upper and lower bounds on $F^{\mathcal{P}}(p, t)$ ensure that, with probability 1, there is a time when process p is up.

probability of this transition is calculated by summing over all labels that lead from c to c', and giving each such label a weight. Hence, we define $P(c) = nf_{trans}(c) \cdot trans(c)$, where $nf_{trans}(c)$ is the normalization factor for $trans(c)$ and $trans(c)(c')$ is defined as

$$\begin{aligned} trans(c)(c') = \sum_{rmsgs, fails} & [\![(c, (rmsgs, fails), c') \in Tr(\mathcal{A}^R)]\!] \cdot \prod_{p,q} ([\![rmsgs(p)(q) \text{ defined}]\!] \cdot \mathcal{N}(q, p, c.n) \\ & + [\![rmsgs(p)(q) \text{ undefined}]\!] \cdot [\![q \notin c.F]\!] \cdot (1 - \mathcal{N}(q, p, c.n))) \\ & \cdot \prod_{p} ((1 - F^{\mathcal{P}}(p,\, c'.n)) \cdot [\![p \in fails]\!] + F^{\mathcal{P}}(p,\, c'.n) \cdot [\![p \notin fails]\!]). \end{aligned}$$

Here, $[\![\cdot]\!]$ maps the Boolean values true and false to 1 and 0 respectively. Note that the only non-determinism in the transitions of $Tr(\mathcal{A}^R)$ comes exactly from the behavior we deem probabilistic: messages being dropped by the network, and process failures.
It is easy to see that $nf_{trans}(c)$ is well defined for all c, as for a fixed configuration c, $trans(c)(c')$ is non-zero for only finitely many configurations c'.

- The distribution over the initial states defined by $\iota = norm(init)$, where
  - $P_f(F) = \prod_{p \notin F} F^{\mathcal{P}}(p, 0) \cdot \prod_{p \in F} (1 - F^{\mathcal{P}}(p, 0))$,
  - $init(n, s, F) = P_f(F) \cdot [\![n = 0]\!] \cdot [\![s \in \mathcal{A}^R.I]\!]$,

  and $norm$ normalizes the probabilities. Note that normalization is possible, since we assumed that after fixing N, each algorithm comes with a finite set of initial states.

Summary of time and failure assumptions

Time. Processes are synchronous and operate in a time-triggered fashion. Links are synchronous (all delivered messages respect a timing upper bound on delivery).
Failures. Processes and links can fail and recover infinitely often. At the beginning of any time-step a crashed process/link can recover with positive probability and a correct process/link can fail with positive probability.

Important: Results in Sections 4 and 5 hold for both crash-recovery models (Section 3.2 and 3.3). Results in Section 6 rely on probabilities.

4 Wrapper for Crash-Stop Algorithms

We now define the transformation of a crash-stop algorithm $\mathcal{A}$ into a crash-recovery algorithm $\mathcal{A}^R$. Intuitively, we do this by (also illustrated in Figure 1):

- Generating a synchronous crash-recovery step using a series of crash-stop steps. Each step in the series handles one individual received message, allowing us to iteratively handle multiple simultaneously incoming messages and bridge the synchrony mismatch between the crash-stop and crash-recovery models.
- Using round-by-round failure detectors to produce the failure detector outputs to be fed to the crash-stop algorithm. These outputs are from the set 2^{Π}.[5]
- Providing reliable links, as required by the crash-stop algorithm. During each crash-recovery step, we buffer all outgoing messages of a process, and send them repeatedly in the subsequent crash-recovery steps, until an acknowledgment is received.

[5] We could instead produce outputs that never suspect anyone, since no process crashes forever in our probabilistic model. However for a weaker model that we define in the full version (where processes are allowed to crash forever), we need failure detectors that suspect processes.

We first define the message and state spaces of the crash-recovery version $\mathcal{A}^R$ of a given crash-stop algorithm $\mathcal{A}$ as follows:

- In $\mathcal{A}^R$, we send a pair of messages to each process in each step: (1) the actual *payload* message (from $\mathcal{A}$), replaced by a special *heartbeat* message *hb* being sent when no payload needs to be sent, and (2) an *acknowledgment* message, confirming the receipt of the last message on the channel in the opposite direction,
- The local state s_p of a process p has three components $(st, buff, acks)$: (1) st stores the state of p in the target crash-stop algorithm; (2) *buff* represents p's outgoing message buffers, with one buffer for each process (including one for p); and (3) $acks(q)$ records the last message that p received from q. The buffers are LIFO, a choice which proves crucial for our termination proof (Section 6).

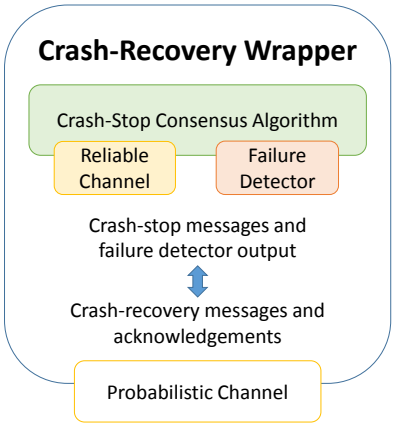

Figure 1 Wrapper concept.

Next, given a crash-recovery state s, a process p, and the messages *rmsgs* received by p in the given round, we define $unfold(p, s, rmsgs)$, p's local step unfolding for s and *rmsgs*. We define $unfold(p, s, rmsgs)$ as the sequence of intermediate steps p takes. Said differently, $unfold(p, s, rmsgs)$ is a sequence of crash-recovery states and crash-stop labels $s_0, l_0, s_1, l_1, s_2, \ldots s_n$, where the intermediate state s_i represents the state of p after processing the message of the i-th process. In a crash-recovery run, p transitions directly from s_0 to s_n. The intermediate states are listed here separately to intuitively show how s_n is computed. The unfolding also allows to relate traces of $\mathcal{A}$ and $\mathcal{A}^R$ more easily in our proofs, as we produce a crash-stop run from a crash-recovery run when proving properties of the wrapper. The content of p's buffers changes as we progress through the states s_i of $unfold(p, s, rmsgs)$, as the wrapper routes the messages to $\mathcal{A}$ and receives new ones from it. The failure detector output (recorded in the labels l_i) remains constant through the unfolding: all processes from whom no message was received in the crash-recovery step are suspected. Finally, the set of failed processes in each label is defined to be empty. We emulate the process recovery that is possible in the crash-recovery model by crash-stop runs in which no processes fail.

Finally, given a crash-stop algorithm $\mathcal{A}$ with an unreliable failure detector and with the per-process state space Σ_p, we define its *crash-recovery version* $\mathcal{A}^R$ where:

- the initial states of $\mathcal{A}^R$ constitute the set of crash-recovery configurations c such that there exists a crash-stop configuration $c_s \in \mathcal{A}.I$ satisfying the following: (i) the initial states of c and c_s correspond to each other and in c all buffers are empty, (ii) no messages are acknowledged, and (iii) the failed processes in c and c_s are the same.
- the next state of a process p is computed by unfolding, based on the messages p received in this round.
- the message that p sends to a process q pairs the first element of p's (LIFO) buffer for q with the acknowledgment for the last message that p received from q.
- the execution is short-circuited as soon as a process decides. This is achieved by broadcasting a message to all other processes, announcing that it has decided. When processes receives such a message, they immediately decide and short-circuits their execution.

Short-circuiting behavior is a common pattern for consensus algorithms [8, 20]. It can be applied in a black-box way and it is sound for any crash-stop consensus algorithm.

A more formal description of the wrapper can be found in the full version. We overload the function symbol $tr_s(\cdot)$ to work on both crash-stop and crash-recovery runs. Given a crash-recovery run ρ^R, we define its *state trace*, $tr_s(\rho^R)$, as the sequence obtained by first removing the labels, then projecting each resulting configuration c onto $c.s$, and finally projecting each local state $c.s_p$ onto $c.s_p.st$. Note that both crash-stop and crash-recovery state properties are sequences of states from the same state space Σ.

5 Preservation Results

As our first main result, we show that crash-recovery versions of algorithms produced by our wrapper preserve a wide class of safety properties. The class includes the safety properties of consensus: validity, integrity and agreement (Section 3). In other words, if a trace of a crash-recovery version of an algorithm violates a property, then some crash-stop trace of the same algorithm also violates that property. We show this in the non-probabilistic crash-recovery model. However, the result also translates to the probabilistic model, since all allowed traces of the probabilistic model are also traces of the non-probabilistic one.

Preserving all safety properties for all algorithms and failure detectors would be too strong of a requirement, for two reasons. First, as our crash-recovery model assumes nothing about link or process reliability, in finite runs we can give no guarantees about the accuracy of the simulated failure detectors. Second, the crash-recovery model is synchronous, meaning that different processes take steps simultaneously. This is impossible in the crash-stop model, which is asynchronous. Thus, the following simple safety property *OneChanges* defined by "the local state of at most one process changes between two successive states in a trace" holds in the crash-stop model, but not in the crash-recovery model (equivalently, we can find a crash-recovery trace, but not a crash-stop trace that violates the property).

We work around the first problem by assuming that the crash-stop algorithms use unreliable failure detectors. For the second problem, we restrict the class of safety properties that we wish to preserve as follows. Consider a property P. Let $u \notin P$ and w be runs with the same initial states (i.e., $u(1) = w(1)$) such that u is a subword of w (recall we define subwords earlier). P then belongs to the class of properties *not repairable through detours* if $w \notin P$ for all such u and w. Intuitively, this means that the sequence of states represented by u inherently violates P; so adding "detours" by the means of additional intermediate states (forming w) does not help satisfy P.

The property *OneChanges* is an example of a safety property that *is* repairable through detours: we can take *any* word that violates *OneChanges* and extend it to a word that does not violate *OneChanges*. However, we can easily show that the following safety properties are not repairable through detours:

- The safety properties of consensus. E.g., consider the validity property: given a word u such that $v = u(i,p).dec$ is a non-initial and a non-$\perp$ value, adding further states between the initial state and $u(i)$ does not change the fact that v is neither initial nor $\perp$.
- *State invariant* properties, defined by a set S of "good" states, such that for a trace u, $u \in Inv(S)$ only if $\forall i.\ u(i) \in S$. Equivalently, these properties rule out traces which reach the "bad" states in the complement of S. Intuitively, if a bad state is reached, we cannot fix it by adding more states before or after the bad state.

We establish the following lemma, which is essential to the ensuing preservation theorem.

▶ **Lemma 2** (Crash-recovery traces have crash-stop superwords)**.** *Let $\mathcal{A}$ be a crash-stop algorithm with an unreliable failure detector. Let ρ^R be a finite run of the crash-recovery wrapper $\mathcal{A}^R$. Then, there exists a finite run ρ of $\mathcal{A}$ such that $tr_s(\rho^R)$ is a subword of $tr_s(\rho)$,*

$tr_s(\rho^R)(1) = tr_s(\rho)(1)$, $last(tr_s(\rho)) = last(tr_s(\rho^R))$, no processes fail in ρ, and in-flight messages of $last(\rho)$ match the messages in the buffers of $last(\rho^R)$.

▶ **Theorem 3** (Preservation of detour-irreparable safety properties)**.** *Let $\mathcal{A}$ be a crash-stop algorithm with an unreliable failure detector, and let P be a safety state property that is not repairable through detours. If $\mathcal{A}$ satisfies P, then so does $\mathcal{A}^R$.*

Proof. We prove the theorem's statement by proving its contrapositive. Assume $\mathcal{A}^R$ violates P. By the definition of safety properties [4], there exists a finite run ρ^R of $\mathcal{A}^R$ such that no continuation of $tr_s(\rho^R)$ is in P. By Lemma 2, there exists a run ρ of $\mathcal{A}$ such that $tr_s(\rho^R)$ is a subword of $tr_s(\rho)$. By Lemma 1, ρ can be extended to an infinite run $\rho \cdot w$ of $\mathcal{A}$. By the choice of ρ^R, we have that $tr_s(\rho^R) \cdot tr_s(w) \notin P$. As $tr_s(\rho^R) \cdot tr_s(w)$ is a subword of $tr_s(\rho \cdot w)$, and since P is not detour repairable, then also $tr_s(\rho \cdot w) \notin P$. Thus, $\mathcal{A}$ also violates P. ◀

Proving that the safety properties of consensus are detour-irrepairable, means that these properties are preserved by our wrapper. Since state invariants are also detour-irreparable, they too are preserved by our wrapper. This makes our wrapper potentially useful for reusing other kinds of crash-stop algorithms in a crash-recovery setting, not just the consensus ones.

▶ **Corollary 4.** *If $\mathcal{A}$ satisfies the safety properties of consensus, then so does $\mathcal{A}^R$.*

▶ **Corollary 5.** *If $\mathcal{A}$ satisfies a state invariant, then so does $\mathcal{A}^R$.*

6 Probabilistic Termination

Termination of consensus algorithms depends on stable periods, during which communication is reliable and no crashes or recoveries occur. In this section, we first state a general result about so-called *selective stable periods* for our probabilistic crash-recovery model. We then define a generic class of crash-stop consensus algorithms, which we call *bounded algorithms*. We prove that termination for these algorithms is guaranteed in our probabilistic model when run under the wrapper. Namely, we prove that, with probability 1, all processes eventually decide. We also show that the class of bounded algorithms covers a wide spectrum of existing algorithms including the celebrated Chandra-Toueg [8] and the instantiation of the indulgent framework of [20] that uses failure detectors.

6.1 Selective Stable Periods

Similar to [11], our proofs will rely on forming strongly-connected communication components between particular sets of processes. However, we will require their existence only for bounded periods of time, which we call selective stable periods.

▶ **Definition 6** (Selective stable period)**.** Fix a crash-recovery algorithm $\mathcal{A}^R$. A *selective-stable period* of $\mathcal{A}^R$ of length Δ for a crash-recovery configuration c and a set of processes C, written $stable(\mathcal{A}^R, \Delta, c, C)$, is the set of all sequences $c = c_0, c_1, \ldots c_{\Delta+1}$ of crash-recovery configurations such that $\forall i.\ 1 \leq i \leq \Delta + 1$ we have $c_i.F = \Pi \setminus C$ and there exist $rmsgs_i$ such that $(c_i, (rmsgs_i, \emptyset), c_{i+1})$ is a step of $\mathcal{A}^R$ and $rmsgs_i(p)(q)$ is defined $\forall p, q \in C$.

Such selective stable periods must occur in runs of a crash-recovery algorithm $\mathcal{A}^R$.

▶ **Lemma 7** (Selective stable periods are mandatory)**.** *Fix a crash-recovery algorithm $\mathcal{A}^R$, a positive integer Δ and a selection function $sel : \mathcal{C}^R \to 2^\Pi$, mapping crash-recovery configurations to process sets. Then, the set of crash-recovery runs*

$$\{c_0, l_0, c_1, \ldots \mid \forall i \geq 0.\ c_i, l_i, \ldots, c_{i+\Delta+1} \notin stable(\mathcal{A}^R, \Delta, c_i, sel(c_i))\}, \text{ has a probability of } 0.$$

The next section shows how our wrapper exploits such periods to construct crash-recovery algorithms from existing crash-stop ones in a blackbox manner. For future work it might also be interesting to devise consensus algorithm directly on top of this property.

6.2 Bounded Algorithms

We next define the class of *bounded* crash-stop algorithms for which our wrapper guarantees termination in the crash-recovery setting. This class comprises algorithms which operate in rounds, with an upper bound on the number of messages exchanged per round as well the number of rounds correct processes can be apart. More formally, they are defined as follows.

▶ **Definition 8** (Bounded algorithms)**.** A crash-stop consensus algorithm (using reliable links and a failure detector [8]) is said to be *bounded* if it satisfies all properties below:

(B1) **Communication-closed rounds:** processes operate in rounds. The rounds must be communication-closed [14]: only the messages from the current round are considered.

(B2) **Externally triggered state changes:** After the first step of every round, the processes change state only upon receipt of round messages, or on a change of the failure detector output.

(B3) **Bounded round messages:** There exists a bound B_s such that, in any round, a process sends at most B_s messages to any other process.

(B4) **Bounded round gap:** Let $N_c = N - N_f$, that is, the number of correct processes according to the algorithm's resilience criterion. Then, there exists a bound B_Δ, such that the fastest N_c processes are at most B_Δ rounds apart.

(B5) **Bounded termination:** There exists a bound B_{adv} such that for any reachable configuration c where any N_c fastest processes in c are correct, the other processes are faulty, and the failure detector output at these processes is perfect after c, then all of these N_c processes decide before any of them reaches the round $r_{\max}(c) + B_{adv}$.

Checking (B1)-(B3) for a given algorithm is typically trivial. We can check (B4) by examining under which condition(s) a process increments its round number, and (B5) by observing the algorithm's termination under perfect failure detection given a quorum of correct processes. Section 6.3 shows an example of these checks. If an algorithm satisfies the Definition 8, we next prove that it terminates in all sufficiently long selective stable periods.

▶ **Theorem 9** (Bounded selective stable period termination)**.** *Let $\mathcal{A}$ be a bounded algorithm and $\mathcal{A}^R$ its wrapped crash-recovery version. Let c be a reachable crash-recovery configuration of the $\mathcal{A}^R$, and let C be some set of N_c fastest processes in c. Then, there exists a bound B, such that, for any selective stable period of length B for c and C, all processes in C decide in $\mathcal{A}^R$. Moreover, the bound B is independent of the configuration c.*

Proof. We partition the processes from C into the set A (initially C) and NA (initially $\emptyset$). The processes in A will advance their rounds further, and the processes in NA will not advance, but will have already decided. Let p be some slowest process from A in c. We first claim that p advances or decides in any selective stable period for c and C of length at most B_{slow}, defined as $B_{slow} = B_s \cdot B_\Delta + 1$. Denote the period configurations by $c = c_0, c_1, \ldots$. Then, using Lemma 2, we obtain a crash-stop configuration c_1^s from c_1 that satisfies the conditions of bounded termination, with the processes from C being correct, and the others faulty. We consider two cases.

First, if p advances in the crash-stop model after receiving all the in-flight round messages in c_1^s from other processes in A, then it also advances in the crash-recovery model after receiving these messages. Moreover, our wrapper delivers all such messages within $B_s \cdot B_\Delta$

steps, as (1) it uses LIFO buffers; (2) bounds B_Δ and B_s apply to c_1^s by (B3) and (B4); and (3) by Lemma 2, the same bounds also apply to c_1 (as (B4) is an invariant).

Second, if p does not advance in the crash-stop model, then, since the failure detector output remains stable, and since no further round messages will be delivered to p in the crash-stop model, the requirement (B2) ensures that p will not advance further in the crash-stop setting. Moreover, requirements (B5) and (B2) ensure that p must decide after receiving all of its round messages; we move p to the set NA.

We have thus established that the slowest process from A can move to either NA or advance its round after B_{slow} steps. Next, we claim that we can repeat this procedure by picking the slowest member of A again. This is because the procedure ensures that the processes in NA always have round numbers lower than the processes in A. Thus, due to (B1) and (B2), the processes in A cannot rely on those from NA for changing their state.

Lastly, we note that this procedure needs to be repeated at most $B_{iter} = N \cdot (B_\Delta + B_{adv})$ times before all processes move to the round $r_{max}(c) + B_{adv}$, by which point (B5) guarantees that all processes terminate. Thus $B_{slow} \cdot B_{iter}$ gives us the required bound B. ◀

The main result of this paper shows that the wrapper guarantees all consensus properties for wrapped bounded algorithms, including termination.

▶ **Theorem 10** (CR consensus preservation). *If a bounded algorithm $\mathcal{A}$ solves consensus in the crash-stop setting, then $\mathcal{A}^R$ also solves consensus in the probabilistic crash-recovery setting.*

Proof. By Corollary 4, we conclude that $\mathcal{A}^R$ solves the safety properties of consensus in the crash-recovery setting. For (probabilistic) termination, the result follows from Theorem 9 and Lemma 7, using as the selection function for Lemma 7 (1) any function that selects some N_c fastest processes in a configuration if no process has decided yet and (2) all processes, if some process has decided. The latter allows us to propagate the decision to all processes, due to short-circuiting in $\mathcal{A}^R$. ◀

6.3 Examples of Bounded Algorithms

We next give two prominent examples of bounded algorithms: the Chandra-Toueg (CT) algorithm [8] and the instantiation of the indulgent framework of [20] that uses failure detectors. For these algorithms, rounds are composite, and consist of the combination of what the authors refer to as rounds and phases. Checking that the algorithms then satisfy conditions (B1)–(B3) is straightforward. (B4) holds for the CT algorithm with $B_\Delta = 4 \cdot N$: take the fastest process p in a crash-stop configuration; if its CT round number r_p is N or less, the claim is immediate. Otherwise, p must have previously moved out of the phase 2 of the last round r'_p in which it was the coordinator, which implies that at least N_c processes have also already executed r_p. Since CT uses the rotating coordinator paradigm, $r_p - r'_p \leq N$; as each round consists of 4 phases, $B_\Delta = 4 \cdot N$. For the algorithm from [20], processes only advance to the next round (which consists of two phases) when they receive messages from N_c other processes. Thus, $B_\Delta = 2$. Finally, proving the requirement (B5) is similar to, but simpler than the original termination proofs for the algorithms, since it only requires termination under conditions which includes perfect failure detector output. For space reasons, we do not provide the full proofs here, but we note that $B_{adv} = \mathit{phases} \cdot \lfloor N/2 \rfloor$ for both algorithms, where *phases* is the number of phases per algorithm round. Intuitively, within this many rounds the execution hits a round where a correct processor is the coordinator; since we assume perfect failure detection for this period, no process will suspect this coordinator, and thus no process will move out of this round without deciding.

▶ **Corollary 11.** *The wrapped versions of Chandra-Toueg's algorithm [8] and the instantiation of the indulgent framework of [20] using failure detectors solve consensus in the probabilistic crash-recovery model.*

7 Concluding Remarks

This paper introduced new system models that closely capture the messy reality of distributed systems. Unlike the usual distributed computing models, we account for failure and recovery patterns of processes and links in a probabilistic and temporary, rather than a deterministic and perpetual manner. Our models allow an unbounded number of processes and communication links to probabilistically fail and recover, potentially for an infinite number of times. We showed how and under which conditions we can reuse existing crash-stop distributed algorithms in our crash-recovery systems. We presented a wrapper that allows crash-stop algorithms to be deployed unchanged in our crash-recovery models. The wrapper preserves the correctness of a wide class of consensus algorithms.

Our work opens several new directions for future investigations. First, we currently model failures of processes as well as communication links individually and independently, with a non-zero probability of failing/recovering at any point in time. In the full version, we sketch how our results can be extended to systems where some processes may never even recover from failure. It is interesting to investigate what results can be established with more complicated probability distributions, e.g., if the model is weakened to allow processes and links to fail/recover on average with some non-zero probability [15]. Second, our wrapper fully persists the processes state. Studying how to minimize the amount of persisted state while still allowing our results (or similar ones) to hold is another promising direction. Finally, we focus on algorithms that depend on the reliability of message delivery. Some algorithms, notably Paxos [23], do not. Finding a modular link abstraction for the crash-stop setting that identifies these algorithms is another interesting topic. For those algorithms, we speculate that preserving termination in the crash-recovery model is simpler.

References

1 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed computing*, 13(2):99–125, 2000.

2 Dan Alistarh, James Aspnes, Valerie King, and Jared Saia. Communication-efficient randomized consensus. In *Distributed Computing*, pages 61–75, 2014.

3 Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. How to solve consensus in the smallest window of synchrony. In *DISC*. Springer, 2008.

4 Bowen Alpern and Fred Schneider. Defining Liveness. *Information Processing Letters*, 21:181–185, June 1985.

5 James Aspnes, Hagit Attiya, and Keren Censor. Combining shared-coin algorithms. *J. Parallel Distrib. Comput.*, 70(3):317–322, 2010.

6 Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

7 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4), 1985.

8 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

9 Bernadette Charron-Bost, Martin Hutle, and Josef Widder. In search of lost time. *Information Processing Letters*, 110(21), 2010.

10 Flavin Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, 1991.

11 Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. In *PODC*, pages 286–, 1997.

12 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

13 Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne Anne Pignolet. Never say never - probabilistic and temporal failure detectors. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 679–688, 2016.

14 Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2:155–173, 1982.

15 Christof Fetzer, Ulrich Schmid, and Martin Susskraut. On the possibility of consensus in asynchronous systems with finite average response times. In *25th IEEE International Conference on Distributed Computing Systems*, pages 271–280, 2005.

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

17 Pierre Fraigniaud, Mika Göös, Amos Korman, Merav Parter, and David Peleg. Randomized distributed decision. *LNCS*, 27, 2014.

18 Felix C. Freiling, Christian Lambertz, and Mila Majster-Cederbaum. Modular Consensus Algorithms for the Crash-Recovery Model. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 287–292, December 2009.

19 Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *PODC*, pages 143–152, 1998.

20 Rachid Guerraoui and Michel Raynal. A generic framework for indulgent consensus. In *ICDCS*, pages 88–, 2003.

21 Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, pages 280–, 1998.

22 Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. A versatile family of consensus protocols based on chandra-toueg's unreliable failure detectors. *IEEE Transactions on Computers*, 51(4):395–408, 2002.

23 Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

24 Mikel Larrea, Cristian Martín, and Iratxe Soraluze. Communication-efficient leader election in crash–recovery systems. *Journal of Systems and Software*, 84(12):2186 – 2195, 2011.

25 Neeraj Mittal, Kuppahalli L. Phaneesh, and Felix C. Freiling. Safe termination detection in an asynchronous distributed system when processes may crash and recover. *Theor. Comput. Sci.*, 410(6-7):614–628, February 2009.

26 H. Moniz, N.F. Neves, and M. Correia. Turquois: Byzantine consensus in wireless ad hoc networks. In *DSN*, 2010.

27 Henrique Moniz, NunoFerreira Neves, Miguel Correia, and Paulo Veríssimo. Randomization can be a healer: Consensus with dynamic omission failures. In *LNCS*, volume 5805. 2009.

28 Rui Oliveira, Rachid Guerraoui, and André Schiper. Consensus in the crash-recover model. 1997.

29 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

30 André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distrib. Comput.*, 10(3):149–157, 1997.

31 Ulrich Schmid, Bettina Weiss, and Idit Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.