Reihe Informatik. TR-2008-002

Easy Consensus Algorithms for the Crash-Recovery Model

Felix C. Freiling, Christian Lambertz, and Mila Majster-Cederbaum

Department of Computer Science, University of Mannheim, Germany

Abstract. In the crash-recovery failure model of asynchronous distributed systems, processes can temporarily stop to execute steps and later restart their computation from a predefined local state. The crash-recovery model is much more realistic than the crash-stop failure model in which processes merely are allowed to stop executing steps. The additional complexity is reflected in the multitude of assumptions and the technical complexity of algorithms which have been developed for that model. We focus on the problem of consensus in the crash-recovery model, but instead of developing completely new algorithms from scratch, our approach aims at reusing existing crash-stop consensus algorithms in a modular way using the abstraction of failure detectors. As a result, we present three new and relatively simple consensus algorithms for the crash-recovery model for different types of assumptions.

1 Introduction

1.1 From Crash-Stop to Crash-Recovery

One of the most popular failure models for fault-tolerant distributed algorithms is called *crash-stop* (or simply *crash*). This model allows that a certain number of processes stops to execute steps forever at some point during the execution of the algorithm. Crash-stop is a very attractive model, especially if it is paired with the *asynchronous message-passing* system model of distributed computations. In this model, processes communicate by sending messages to each other, however, neither the message delivery delay nor the relative processing speeds of processes are bounded. It is well-known that in such systems many important algorithmical problems are unsolvable, for example *consensus* [5].

Despite its theoretical attractiveness, the crash-stop model is not expressive enough to model many realistic scenarios. In practice, processes crash but their processors reboot and the crashed process is restarted from a recovery point and rejoins the computation. This behavior is especially common for long-lived applications like distributed operating systems, grid computing or web services and has been formalized as a failure model called *crash-recovery*. In the crash-recovery model, processes can crash multiple times. After crashing (and before crashing the next time), a process recovers from a predefined state.

Crash-recovery is a strict generalization of crash-stop, i.e., every faulty behavior allowed in crash-stop is also possible in crash-recovery. This means that any impossibility result for the crash-stop model also holds in the crash-recovery model. However, algorithms designed for the crash-stop model will not necessarily work in the crash-recovery model due to the additional faulty behavior. This additional behavior is surprisingly complex. For example, while in the crash-stop model processes are usually classified into two distinct classes (those which eventually crash and those which do not), in the crash-recovery model we already have four distinct classes of processes: (1) *always up* (processes that never crash), (2) *eventually up* (processes that crash at least once but eventually recover and do not crash anymore), (3) *eventually down* (processes that crash at least once and eventually do not recover anymore), and (4) *unstable* (processes that crash and recover infinitely often). As another example of increased complexity, processes in the crash-recovery model usually lose all state information when they crash. The notion of *stable storage* was invented to model a type of expensive persistent storage which is usually available in practice in the form of hard disks.

1.2 Consensus in the Crash-Recovery Model

The additional expressiveness of the crash-recovery model has a price that can be estimated when looking at distributed algorithms for this model. In this paper, we choose the problem of *distributed consensus* as benchmark problem to study the differences between crash-stop and crash-recovery. Roughly speaking, consensus requires that a set of processes in a distributed system have to agree on a common value from a set of input (or *proposal*) values from each process. Consensus is fundamental to many fault-tolerant synchronization problems but has mainly been studied in the crash-stop model. As mentioned above, it is even impossible to solve deterministically in asynchronous systems [5], but becomes solvable using extensions of the model. In this paper, we use the *failure detector* abstraction [3] to solve consensus in the crash-recovery model. Intuitively, a failure detector is a distributed oracle that gives information about the failures of other processes. We look at two classes of such failure detectors in this paper: the class of *perfect failure detectors* ($\mathcal{P}$) that tell exactly who has failed, and the class of *eventually perfect failure detectors* ($\Diamond\mathcal{P}$) that can make finitely many mistakes in telling the failure state of other processes (i.e., they *eventually* behave like perfect failure detectors).

Compared with the crash-stop model, consensus algorithms in the crash-recovery model have to deal with several problems. The first problem is: How do we deal with recovering processes? Recovering processes have to be re-integrated into the computation so that they can terminate the protocol in a well-defined way. For example, if they already *had* terminated the protocol with a certain decision value v, then the algorithm must ensure that they never terminate with a decision value which is not v in the future.

The second problem we have to deal with is: How do we deal with unstable processes? In unfavourable circumstances, unstable processes can cause algorithms to run forever. An unstable process can crash, then recover, then crash, then recover, infinitely often. Between each recovery and subsequent crash, the process can upset and delay the decision making process by requesting a state update or proposing a new decision value. It is this problem that lead to the definition of *quiescence* of algorithms. Intuitively, an algorithm is quiescent if it eventually stops to send messages. Obviously, consensus algorithms in the crash-recovery model can only be quiescent if there are no unstable processes.

Finally, the third problem to handle in crash-recovery is: How to deal with messages sent to processes which are crashed but later recover? In the crash-stop model, communication channels were usually assumed to be reliable, but in crash-recovery, message loss is a natural effect which increases the complexity. Most often, this problem is dealt with using the abstraction of *stubborn communication channels* [6]. Briefly spoken, such channels infinitely often re-send a message as long as the sender does not fail. Therefore, a message sent over a stubborn channel will eventually be received at its destination as long as the receiver eventually recovers.

1.3 Related Work

The most recent work on consensus in the crash-recovery model was published by Aguilera, Chen and Toueg [2]. They introduce the four classes of processes mentioned above and prove necessary conditions to solve consensus without stable storage. They also give a necessary condition for the case with stable storage assuming that only the proposal and decision value may be saved. For this condition they give a rather complex algorithm. The condition states that more always up than incorrect—eventually down and unstable—processes must be present, if only an eventually perfect failure detector is available. If more information is allowed to be saved on stable storage, a further intricate algorithm is constructed which assumes the presence of a majority of correct processes.

Oliveira, Guerraoui, and Schiper [11] also give an interesting consensus algorithm for the crash-recovery model, but in their model the processes do not lose any state information due to crashes. Hurfin, Mostéfaoui and Raynal [8] developed a complex voting based consensus algorithm that uses stable storage. Both papers [8, 11] use a failure detector definition that was later shown [2] to exhibit anomalous behavior: The definition allows runs in which correct processes eventually permanently trust unstable processes. We avoid such behavior in this paper.

Although mentioning neither crash-recovery nor failure detectors, Lamport's Paxos algorithm [9] also solves consensus in the setting of this paper. However, like Aguilera, Chen and Toueg [2], Paxos is also rather intricate and complex. While re-using algorithmical ideas from the area of crash-stop algorithms, all previously mentioned algorithms were built from scratch.

1.4 Contributions

Similarly to Aguilera, Chen and Toueg [2] we investigate the solvability of consensus in the crash-recovery model under varying assumptions. However, our approach is to re-use existing algorithms from the crash-stop failure model in a modular way. One main task of our algorithms therefore is to partly *emulate* a crash-stop system on top of a crash-recovery system to be able to run a crash-stop consensus algorithm. In this sense, our approach can also be regarded as a *transformation* of a crash-recovery system into a crash-stop system.

Table 1 summarizes the cases we study in this paper along three dimensions: (1) the availability of stable storage (large columns left and right), (2) a process state assumption (rows of the table), and (3) the availability of failure detectors (sub-columns within large columns). Impossibility results are denoted by "$\times$" and solvability by "✓". Impossibility results with stronger assumptions imply impossibility for cases with weaker assumptions, while solvability with weak assumptions implies solutions with stronger assumptions. We have ordered the strength of the parameters so that they are increased from left to right and from top to bottom.

assumptions	**no stable storage**		**stable storage**	
	$\Diamond\mathcal{P}$	$\mathcal{P}$	$\Diamond\mathcal{P}$	$\mathcal{P}$
one correct	$\times$	$\times$	$\times$	$\times$ Lemma 2
correct majority	$\times$	$\times$ Sect. 4.1	✓ Theorem 2	✓
one always up	$\times$	✓ Theorem 1	$\times$ Lemma 1	✓ Theorem 1
correct majority & one always up	$\times$ [2]	✓	✓	✓
more always up than incorrect	✓ [2]	✓	✓ [2]	✓
always up majority	✓ Sect. 4.3	✓	✓ Sect. 4.3	✓

Table 1: Overview of the results of this paper. An arrow that connects two cells depicts a logical implication.

As mentioned before, the case of $\Diamond\mathcal{P}$ and more always up than incorrect processes was proven to be the weakest for consensus [2] and thus, all weaker process state assumptions are impossible. We first focus on $\mathcal{P}$ and the unavailability of stable storage. We argue in Sect. 4.1 that at least one always up process is necessary and sufficient. The sufficiency part is proved in Theorem 1. Therefore, consensus is also solvable under stronger process asumptions. Then we weaken $\mathcal{P}$ to $\Diamond\mathcal{P}$ and present a modular algorithm for the always up majority of processes case in Sect. 4.3. This completes the discussion for the case with no stable storage.

We then turn to the case where processes are allowed to use stable storage. We first prove two impossibility results regarding the presence of correct processes. The first impossibility arises in the case where we have only an eventually perfect failure detector and one always up process (Lemma 1). The second impossibility arises in the case where the perfect failure detector is available and one eventually up process is present (Lemma 2). We then give an algorithm for the remaining case (see Theorem 2): It uses the eventually perfect failure detector, a majority of always up or eventually up processes, and some minimal insight about the used crash-stop consensus algorithm which needs to be saved on stable storage.

In all studied cases of this paper, the aim of the research is simplicity over efficiency, i.e., the solutions rather avoid performance improvements if they inhibit comprehensibility, although some performance optimizations are obvious and could easily be added, e.g., we use failure detectors with strong accuracy properties instead of detectors with weak accuracy properties. Thus, similar to the easy impossibility proofs for consensus [4], we develop easy consensus algorithms.

1.5 Roadmap

We first present all necessary definitions in Sect. 2. The basic idea of this paper, the *emulator*, is explained in Sect. 3 and afterwards, we study the cases of Tab. 1 in Sect. 4 and 5. For lack of space, we relegate pseudocode of some algorithms and parts of the proofs to an appendix.

2 System Model

Asynchronous Systems and Failure Models. A distributed system consists of a set of n processes, denoted by $\Pi = \{1, \ldots, n\}$, that is interconnected by a communication network with a fully-connected topology. The system is *asynchronous*, i.e., there is neither a bound on the message delivery delay nor on relative processing speed differences. This means that while one process takes a step, another process can take any finite number of steps.

In the system there exists a discrete global clock, $\mathcal{T} \subseteq \mathbb{N}$. Every step in the system corresponds to a tick, $t \in \mathcal{T}$, of this clock. The processes do not have access to the clock; it just simplifies the presentation of the system.

We assume the failure model of *crash-recovery* in this paper. In this model, processes can fail by crashing. If they crash, they may later recover. A currently crashed process is said to be *down.* A process that is not down is called *up.* We formalize failures by defining a *failure pattern* F as a function that determines the set of processes that are down at a particular point in time. Thus, $\mathrm{F}\colon \mathcal{T} \to 2^{\Pi}$ and $\mathrm{F}(t) = \{p \in \Pi \mid p \text{ is down at } t\}$. If $p \notin \mathrm{F}(t-1)$ and $p \in \mathrm{F}(t)$, p *crashes* at time t. If $p \in \mathrm{F}(t-1)$ and $p \notin \mathrm{F}(t)$, p *recovers* at time t.

Based on the failure pattern, we distinguish four groups of processes:

- *Always up* processes that never crash.
- *Eventually up* processes that crash and recover finitely often, but eventually remain up.
- *Eventually down* processes that crash and recover finitely often, but eventually remain down.
- *Unstable* processes that crash and recover infinitely often.

Always up and eventually up processes together are called *correct* or *good* processes and denoted by $\mathrm{good}(\mathrm{F})$. Similarly, the eventually down and unstable processes are called *incorrect* or *bad* processes, denoted by $\mathrm{bad}(\mathrm{F})$.

The crash-recovery model is a generalization of the *crash-stop model.* In the crash-stop model, processes fail by crashing only, i.e., they never recover; all other definitions are the same as in the crash-recovery model.

If a process crashes and later recovers, the process loses the contents of all variables due to the crash. Processes have the possibility to save variables on *stable storage*, a persistent storage that is preserved during a crash period. Access to stable storage is typically regarded as very expensive and should be minimized, i.e., the number of accesses as well as the amount of stored data should be as small as possible.

Algorithms, Problems and Interfaces. An *algorithm* in a distributed system is a set of n automata, one for each process. These automata proceed in atomic *steps*, which change the *state*—a representation of the local variables—of each process. Every step corresponds to a tick $t \in \mathcal{T}$ of the global clock.

According to a failure pattern F, the automaton at a process p takes only a step—a so-called *normal step*—at a time t, if $p \notin \mathrm{F}(t)$. Otherwise, p is in a special state at time t, the *crash state*. In this state, all local variables not on stable storage are lost. If a process p crashes at a time t, p takes a so-called *crash step* at time $t-1$. If p recovers at time t, it takes a special *recovery step* first and then can take a normal step again at time $t+1$. In one normal step a process can perform a state transition plus any *two* of the following six actions:

- Send/receive a message to/from the network.
- Set/get an external output/input.
- Save/load values of variables on/from stable storage.

A step is sometimes also called an *event*. Events in which external outputs and inputs are affected are called *interface events*. A *problem* is characterized by its interface events and described by a set of properties. Following the ideas of Guerraoui and Rodrigues [7], the *interface* of a problem determines the types of *events* to processes. We distinguish two types of events:

- *Request event*: The algorithm has to handle a request of its caller.
- *Indication event*: The algorithm informs its caller about the event.

An algorithm *implements* a problem if it handles all request events and triggers indication events such that every execution of the algorithm satisfies the properties of the problem.

In the crash-recovery model, every algorithm must handle the following four events:

- Request **init**: Initialize variables at the beginning of the computation.
- Request **recover**: Handle recoveries of crashed processes.
- Indication **save** variables $v_1, \ldots, v_i$ **at** register LOCATION: Save $v_1, \ldots, v_i$ on stable storage at LOCATION.
- Indication **load** variables $v_1, \ldots, v_j$ **at** register LOCATION: Load data into $v_1, \ldots, v_j$ from stable storage at LOCATION.

In this paper we focus on *quiescent algorithms* [1]. An algorithm is *quiescent*, if eventually all processes stop sending messages.

Communication and Failure Detection. The processes in an asynchronous system are connected by *stubborn links* with unbounded message delivery delay.

Stubborn links provide the following properties:

- **No Creation:** If a process q receives a message m from a process p, then m was previously sent to q by p.
- **Stubbornness:** If a process p sends a message m to a good process q, and does not crash, and indefinitely delays sending any further message to q, then q eventually receives m.

Note that the requirement of the indefinite delay does not mean that a process p is *not allowed* to send any new message m' after the sending of a message m in order to ensure the reception of m. In fact, there exists no fixed delay after which process p can be sure that m has been received by the destination process. Instead, stubborn links model an implicit retransmission mechanism allowing to wait for an acknowledgement before sending the next message. Therefore, to achieve quiescence of the stubborn link implementation, we provide a special event to stop this implicit retransmission (this event is called “finalsend” in the original definition of stubborn links [6]).

Stubborn links are a generalization of *reliable links* which are usually used in the crash-stop model. Reliable links guarantee that messages sent to good processes are received exactly once. They do not need to model implicit retransmissions.

The set of all messages is denoted by $\mathcal{M}$ and every message $m \in \mathcal{M}$ is labeled with a unique identifier, which is provided by the system automatically.

If an algorithm wants to use stubborn links, it has to implement the following indication event and can use the following request events:

- Request **send** message m **to** process q: Send m to q stubbornly.

- Indication **receive** message m **from** process q: Receive messages.
- Request **single-send** message m **to** process q: Send m to q once.
- Request **stop-retransmit**: Stop the stubborn sending of any message.

A *failure detector* D is a function $\mathrm{D}\colon \mathit{\Pi} \times \mathcal{T} \rightarrow 2^{\mathit{\Pi}}$ and each process $p \in \mathit{\Pi}$ has access to the value of D. We say that the failure detector at process p *suspects* process q of being down at time t, if $q \in \mathrm{D}(p, t)$. If an algorithm wants to use a failure detector, it has to implement the following indication event:

- Indication **suspect** set of processes Q: Suspicion of all processes $q \in Q$.

We define two classes of failure detectors whose properties are similar to the original failure detector definitions [3]. Note that the strong completeness property has been modified [10] in order to fit the crash-recovery model. The eventually up completeness property is also needed in the crash-recovery model in order to avoid suspicion of eventually up processes when they remain up. Of course, both completeness properties are also valid in the crash-stop model.

A failure detector D belongs to the class $\mathcal{P}$ of *perfect failure detectors*, if it satisfies the following properties:

- **Strong Completeness:** Every bad process is suspected infinitely often by every good process.
- **Eventually Up Completeness:** Eventually, every good process is not suspected any longer.
- **Strong Accuracy:** No process is suspected before it crashes.

The strong accuracy property of the perfect failure detector does not allow any false suspicions, i.e., a process is suspected, but did not crash. In real world implementations this restriction is problematic, because it is hard to distinguish between a slow and a crashed process. Thus, the following definition weakens the accuracy assumption and results in a new failure detector class.

A failure detector D belongs to the class $\Diamond\mathcal{P}$ of *eventually perfect failure detectors*, if it satisfies the following properties:

- **Strong Completeness** and **Eventually Up Completeness**.
- **Eventually Strong Accuracy:** Eventually, no process is suspected before it crashes.

The Consensus Problem. In the *uniform consensus problem* each process *proposes* a value which it obtained initially. Then all processes have to *decide* the same value, i.e., they must agree on one of the proposals and output it. Thus, the interface of uniform consensus consists of the following events:

- Request **propose** value v: Propose v for consensus.
- Indication **decide** value v: Indicates the decision of v.

An algorithm implementing uniform consensus has to satisfy the following properties:

- **Termination:** Every good process eventually decides.
- **Uniform Agreement:** Processes do not decide different values.
- **Validity:** If a process decides value v, then v was proposed by some process.

3 The Emulation Technique and its Limitations

3.1 The Emulator Idea

The aim of this paper is, for different assumptions, to construct a consensus algorithm A_{CR} in the crash-recovery model by using a consensus algorithm A_{CS} from the crash-stop model as a building block. Our basic assumptions about A_{CS} are that it uses a failure detector (either $\mathcal{P}$ or $\Diamond\mathcal{P}$) as synchrony abstraction and reliable or stubborn links as message passing abstraction. Of course, we assume that A_{CS} also solves consensus in the crash-stop model.

We now describe the interface of an *emulator* which simulates a crash-stop failure model on top of a crash-recovery system. The idea is depicted in Fig. 1. At the top of the figure we see the interface of an arbitrary crash-stop consensus algorithm A_{CS}. It consists of the following indication events and request events:

- Request **CS-propose** value v: Propose v for A_{CS}.
- Indication **CS-decide** value v: A_{CS} indicates the decision of v.

- Request **CS-suspect** set of processes Q: Inform A_{CS} that all processes $q \in Q$ are suspected.
- Indication **CS-send** message m **to** process q: A_{CS} wants to send m to q stubbornly.
- Request **CS-receive** message m **from** process q: A_{CS} should receive m sent by q.
- Indication **CS-single-send** message m **to** process q: A_{CS} wants to send m to q once.
- Indication **CS-stop-retransmit**: A_{CS} wants to stop the stubborn sending of all messages.

The available resources within the crash-recovery model are depicted at the bottom of Fig. 1. There we have the failure detector (of class $\mathcal{P}$ or $\Diamond\mathcal{P}$) and a stubborn communication channel abstraction. The emulator is a distributed algorithm inbetween both layers. It must map request and indication events to each other such that A_{CS} together with the emulator solve consensus in the crash-recovery model. Hence, A_{CS} together with the emulator result in A_{CR}. Note that the emulator must also handle the events init, recover, save and load (as described in the system model), which are not depicted in Fig. 1 since they either do not directly affect A_{CS} (recover, save, load) or are internal to A_{CS} (init).

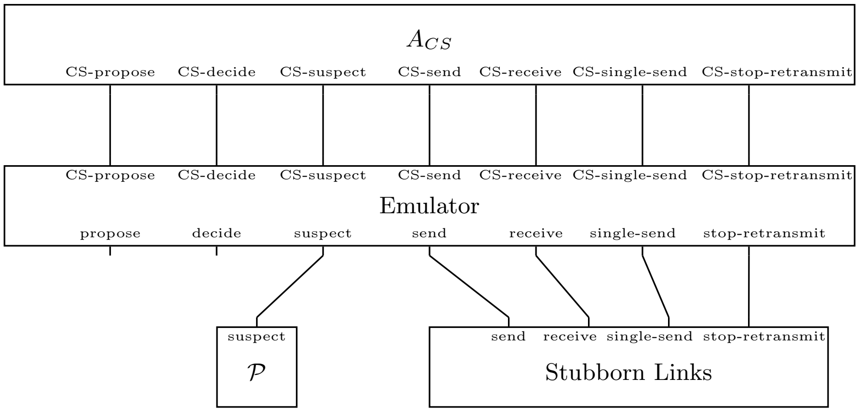

Fig. 1: An emulator algorithm and its connection to the A_{CS} interface.

3.2 Limitations of Modular Solutions

Aguilera, Chen and Toueg [2] proved a necessary and sufficient requirement for solvability of consensus in the crash-recovery model. Without stable storage and only equipped with an eventually perfect failure detector there must be more always up than incorrect processes in the system. Note that this implies that there must be at least one always up process in the system. We now argue, that we cannot employ the emulation technique in these cases.

Any known eventually perfect failure detector based crash-stop consensus algorithms requires the presence of a majority of crash-stop correct processes in the system. But, the new condition in the crash-recovery model of more always up than incorrect processes allows, that a majority of processes crashes finitely often. Thus, the crash-stop algorithm running in the crash-recovery model cannot rely on an always up majority as actually required. A potential consensus algorithm needs to determine the set of currently up processes in each round and rely on this set to guarantee uniform agreement. But, this determination can only be handled by a “non-modular” algorithm, because an emulator cannot influence the number of processes for which the crash-stop consensus algorithm needs to wait in individual rounds. Note that Aguilera, Chen and Toueg [2] present such a (non-modular) algorithm.

4 Easy Consensus Algorithms Without Stable Storage

We now study modular consensus algorithms for the crash-recovery model under the assumption that *no* stable storage is used.

4.1 Necessary Condition Without Stable Storage

The minimal number of correct processes that have to be present in the system to solve consensus in the absence of stable storage is one always up process. To see this, assume that all processes are allowed to crash at least once. In this case all crashes could happen simultaneously. But, in the absence of stable storage a simultaneous crash leads to a total loss of information in the system, because all proposed values are lost. Therefore, the processes have no way to decide. No failure detector is able to prevent this total loss. We give an algorithm using the perfect failure detector in Sect. 4.2, thereby showing that at least one always up process is a necessary and sufficient condition in this setting.

As in the crash-stop model, the availability of only an eventually perfect failure detector requires stronger assumptions for consensus to be solvable. In the crash-stop model, a majority of correct processes (i.e., a majority of processes that never crash) is commonly assumed (e.g., in the original paper by Chandra and Toueg [3]). These can preserve variables and cope with false suspicions of the eventually perfect failure detector. However, in the crash-recovery model the assumption of a majority of correct processes allows that all processes of this majority crash several times as long as they remain up eventually. Even the presence of one always up process and a majority of correct processes cannot preserve a future decision value through asynchronous rounds in a consensus algorithm without stable storage, because the majority of processes can forget the decision estimate due to crashes [2].

4.2 Modular Algorithm based on $\mathcal{P}$

Assuming at least one always up process, we now present an easy consensus algorithm using a perfect failure detector. Any failure detector based quiescent crash-stop consensus algorithm that requires a perfect failure detector and works with at least one crash-stop correct process can be used as A_{CS} in the transformation.

The main idea of the algorithm is to exclude recovered processes from the computation of the crash-stop consensus algorithm. This is achieved by collecting recovered processes together with the crashed processes in a set $Suspected_p$ and by requesting that the crash-stop consensus algorithm should suspect all these processes. Note that recovered processes can be identified by special "I recovered" messages which they broadcast when they recover. Because uniform consensus requires that the eventually up processes among those which are excluded from A_{CS} also decide, special decision messages are broadcast after the decision of the first process. Note that all other events besides the decision event are relayed by the emulator, e.g., if A_{CS} wants to send a message.

The processes stop the stubborn sending of their last message once they decided. This is important in order to guarantee quiescence. Equally important for quiescence is that "I recovered" messages are answered in a non-stubborn fashion.

Theorem 1: Given an asynchronous crash-recovery system with stubborn links, a perfect failure detector, at least one always up process, and a quiescent crash-stop consensus algorithm. Then the above algorithm (depicted as Algorithm 2 in Appendix A) implements uniform consensus.

The proof is also presented in Appendix A.

4.3 Modular Algorithm based on $\Diamond\mathcal{P}$

We now turn to the case where in the absence of stable storage only an eventually perfect failure detector is available. Our solution assumes a majority of always up processes to be present in the system. Any failure detector based quiescent crash-stop consensus algorithm that requires at least an eventually perfect failure detector and works with a majority of crash-stop correct processes can be used as A_{CS} in the transformation.

The idea of the algorithm is similar to the algorithm based on the perfect failure detector. The main difference is the handling of the false suspicions made by the eventually perfect failure detector. In order to separate the possibly falsely suspected processes and the recovered ones, two sets are defined: $\mathit{Suspected}_p$ and $\mathit{Recovered}_p$. The union of these two sets is used as the input for the failure detector of the crash-stop consensus algorithm. Note that the set $\mathit{Suspected}_p$ is updated with any change in the input of the failure detector. The separation of the suspected and recovered processes is important in order to determine the definite state of the processes. The information about the recovery of processes and the possibly false suspicions are not mixed. Thus, no information is lost.

The correctness proof of this algorithm is similar to the proof of Theorem 1 and thus omitted. Interestingly, the always up majority assumption is only needed until A_{CS} terminates. After the decision of the A_{CS} algorithm happened, the emulator needs only at least one always up process in order to disseminate the decision value.

4.4 Transformation Complexity

Both presented algorithms do not increase the complexity of the used algorithm much. Assume that A_{CS} needs m_{cs} messages and r_{cs} rounds to find a decision. If no recoveries occur, at most $|\text{good(F)}| \cdot |\Pi|$ additional decision messages—every correct process broadcasts the decision—are sent and only one more round is needed. Since typically $m_{cs} > |\text{good(F)}| \cdot |\Pi|$, we obtain the same bounds.

With every recovery $|\Pi| + |\text{good(F)}|$ additional messages are sent after the decision already occurred. These are the broadcast of the recovered process plus the answers of every correct process. Before the decision happens, typically only $|\Pi|$ messages are sent by the recovered process and the number of rounds is only increased if, for example, the recovered process was the leader of the round before its recovery (for consensus algorithms based on the rotating coordinator paradigm).

All messages are only altered by a constant number of bits in order to distinguish messages from A_{CS} and the new recovery and decision messages, which in turn have a constant length.

5 Easy Consensus Algorithms Using Stable Storage

Aguilera, Chen and Toueg [2] proved that if stable storage is available but the processes are only allowed to save the proposals and the decision values there, consensus is still not solvable. Thus, more information needs to be saved on stable storage in order to guarantee uniform agreement of consensus. In order to overcome this impossibility, our algorithms are allowed to use stable storage to save important variables.

We now study consensus algorithms for the crash-recovery algorithm in case stable storage is available. Again, we wish to employ the emulation approach. In the emulation approach, however, the information that may be stored on stable storage is restricted to the proposal, the messages, and the decision value. The simplest idea is to save all available information, i.e., the proposal, all messages, and the decision value of each process, on stable storage. But since access to stable storage is expensive, this is not very elegant. In Sect. 5.2 we provide a solution that only stores a subset of messages on stable storage and uses an eventually perfect failure detector. In contrast to our algorithm without stable storage, we merely need a majority of *correct* processes if stable storage is available, not a majority of always up processes. This requirement is minimal, as we show in Sect. 5.1.

Note that we do not investigate the case using a perfect failure detector here, because in this case consensus is solvable with at least one always up process even without stable storage (see Sect. 4.2). We prove that this is a minimal requirement also for systems with stable storage in the following section.

5.1 Necessary Requirements

The new possibilities with stable storage only help to cope with information loss due to recoveries. An eventually perfect failure detector still requires a majority of correct processes in order to cope with false suspicions.

Lemma 1: Consensus is not solvable in asynchronous crash-recovery systems if an eventually perfect failure detector and stable storage are accessible by the processes and if only at least one always up process is assumed to be present in the system.

Proof (Sketch): Assume that such an algorithm is possible and consider a run, in which a process p_1 proposes a value v and immediately after its proposal stops taking any steps until a time t_1. Another process p_2 proposes a value $w \neq v$ and is unable to communicate with p_1, because of the temporary existence of a network problem. The eventually perfect failure detector at p_2 now suspects p_1, and thus p_2 is the only correct process in the system—in its view—and decides w before time t_1. Then p_2 stops taking steps. After time t_1, p_1 decides w in an analogues way, because its failure detector suspects p_2. But, this second decision violates the uniform agreement property of consensus. ■

Lemma 1 highlights that the impossibility of consensus is independent of the availability of stable storage, if a failure detector is present that is prone to false suspicions (like an eventually perfect failure detector). Thus, if such a failure detector is given, the presence of at least a majority of correct processes has to be assumed.

Another problem occurs if stable storage and a perfect failure detector are accessible by the processes, but only at least one correct process is assumed to be present in the system. Interestingly, consensus is not solvable under this assumption, even if the processes can use stable storage for their *entire* state information and a *very perfect* failure detector is given, namely one which *immediately* outputs the exact process state—up or down—at any time it changes.

Lemma 2: Consensus in not solvable in asynchronous crash-recovery systems if a very perfect failure detector and stable storage are accessible by the processes and if only at least one correct process is assumed to be present in the system.

Proof (Sketch): Assume that such an algorithm is possible and consider a run, in which a process p_1 proposes a value v and immediately after its proposal crashes, i.e., the algorithm can only save the proposal on stable storage. Another process p_2 proposes a value $w \neq v$, and the very perfect failure detector at p_2 suspects p_1. Now, p_2 is the only correct process in the system—in its view—and decides w. After its decision, p_2 crashes. Process p_1 recovers after all messages in transit are lost, because they could not be delivered since no process was up. The very perfect failure detector at p_1 suspects p_2, and thus p_1 can only decide v, since v is the only value p_1 knows and—in its view—p_1 is the only correct process in the system. But, this contradicts the uniform agreement property of consensus. ■

5.2 Modular Algorithm based on $\Diamond\mathcal{P}$

Recall that it is not sufficient to store *only* proposal and decision values on stable storage. We now propose an algorithm that uses stable storage to save at every process the *last* message received from any other process and the *last* message that was sent. Roughly speaking, the emulation algorithm extracts the state of the used crash-stop consensus algorithm A_{CS} from these last messages. The temporal term “last” refers to a specific order, as we now explain.

Since the idea of last message storage and the mentioned order strongly depend on the used crash-stop consensus algorithm, we use a concrete crash-stop consensus algorithm as an example. Because it is so well known, we use the original Chandra/Toueg rotating coordinator consensus algorithm adapted to use $\Diamond\mathcal{P}$ [3]. This algorithm, which we call the CT algorithm, is depicted as Algorithm 3 in Appendix B.

The main idea of the CT algorithm is that the processes pass through consecutive rounds as long as the problem is unsolved and in each round one process is the round leader. Because the round number grows larger than the number of processes, the leader is determined by the current round number modulo the number of processes. In every round, the leader tries to impose its current decision estimate among a majority of processes, and since the algorithm runs in the crash-stop model, this majority never crashes. Thereby, it always chooses the freshest estimate from prior rounds as current estimate. Uniform agreement is satisfied, because the first decision happens only after a majority acknowledged the leader's estimate. For more details see Chandra and Toueg [3].

Fig. 2 contains all messages that are sent by the CT algorithm and their purpose. These messages have different importance for a successful run of the algorithm, e.g., a message of type m_4, which contains a potential future decision value, is more important than a new round message of type m_2. Thus, the following order is defined on the messages of Algorithm 3. A message m precedes a message m'—denoted by $m \prec m'$—if m is placed before m' in the following order:

$\langle round_p, \textsc{wakeup}\rangle \prec \langle round_p, \textsc{newround}\rangle$

$\prec \langle round_p, \textsc{estimate}, estimate_p, adopted_p\rangle \prec \langle round_p, \textsc{adopt}, estimate_p\rangle$

$\prec \langle round_p, \textsc{ack}\rangle \prec \langle \textsc{decide}, estimate_p\rangle$

No.	Message	Purpose
m_1	$\langle round_p, \textsc{wakeup}\rangle$	The processes inform the leader of round $round_p$ about a new round.
m_2	$\langle round_p, \textsc{newround}\rangle$	The leader informs about the new round $round_p$.
m_3	$\langle round_p, \textsc{estimate}, estimate_p, adopted_p\rangle$	The processes inform the leader of round $round_p$ that their current decision estimate is $estimate_p$, which was adopted in round $adopted_p$.
m_4	$\langle round_p, \textsc{adopt}, estimate_p\rangle$	The leader of round $round_p$ chooses value $estimate_p$ as the freshest one and informs the other processes.
m_5	$\langle round_p, \textsc{ack}\rangle$	The processes inform the leader of round $round_p$ that its estimate was adopted.
m_6	$\langle \textsc{decide}, estimate_p\rangle$	Value $estimate_p$ was decided; it is safe to decide now.

Fig. 2: A summary of all messages that are sent by the CT algorithm (Algorithm 3).

Messages of the same type, e.g., both m_4, thereby have an additional order, higher round numbers succeed lower ones. If a process received or sent no message so far, a variable for comparison is empty, i.e., it has the value $\bot$. Thus, this message value should also be the lowest in the order.

Since this order depends on the used crash-stop consensus algorithm, it is assumed for simplicity that every crash-stop consensus algorithm defines a suitable order. The emulation algorithm can then use the $\prec$ operator to compare incoming and outgoing messages.

The idea for an emulation algorithm is the saving of the last message that was sent and the last message that was received on stable storage. If a process recovers, it first loads the last messages and its proposal. Then it restarts the A_{CS} algorithm, but directly delivers the last received message and sends out the last sent message.

The delivery of the last received message after a recovery of a previously crashed process should restore the state of the process before it crashed, especially if this state information is essential for the run of the crash-stop consensus algorithm. In the case that Algorithm 3 is used as A_{CS}, the most essential state is the successful adoption of a future decision value, i.e., when a process received message m_4 from the current round leader and sends back acknowledgment message m_5. This situation means that the process agrees to the decision of a certain value. Thus, this state information must be remembered in order to avoid the violation of uniform agreement after a recovery.

The necessary adjustments to an emulation algorithm in order to save the last message information on stable storage are presented in Algorithm 1. This algorithm is based on the emulator algorithm presented in Sect. 4.3. The important difference is the new assumption that a majority of correct processes is needed and not a majority of always up processes as in the case of the algorithm of Sect. 4.3.

If in Algorithm 1 the used crash-stop consensus algorithm of a process sends or receives a message, the emulator compares this message with the most important previously sent or received message. If the new message is more important in the predefined order, it is saved on stable storage as the most important message for future comparisons.

If a process recovers, the emulator tries to load the decision value from stable storage first. If a decision already happened before the crash of the process, it was saved and can now be retrieved. Thus, the process can decide again. Otherwise, the emulator loads the proposal, the last sent message, and the last received message from stable storage. The crash-stop consensus algorithm is started from scratch with the loaded proposal, and the last received message is delivered again, if the corresponding process received one before it crashed. This re-delivery restores the last decision estimate and its adoption time as it was before the crash. Therefore, uniform agreement can be guaranteed in the remaining consensus computation.

The last sent message is also sent again by the emulator. If the process sent no message before it crashed—i.e., the last sent message variable has the value $\perp$—the emulator broadcasts value $\perp$ as last sent message. This is important, because if the other processes decided during the down period of the recovered process, they already stopped sending any message in order to satisfy quiescence. But, the recovered process needs to get to know the decision value somehow and to inform the others of its recovery. Thus, it sends this empty message, and if a process which already decided receives it, it responds with the decision value. The empty message is chosen for this purpose, since it is the lowest message in the message order of crash-stop consensus Algorithm 3—it should also be the lowest in any order—and does not influence any other already stored message.

Theorem 2 (Correctness of Algorithm 1): Assume an asynchronous crash-recovery system with stubborn links, an eventually perfect failure detector, and a majority of correct processes. Given also a quiescent crash-stop consensus algorithm A_{CS} and the relation $\prec$ on messages of A_{CS}. Then Algorithm 1 implements uniform consensus.

Proof (Theorem 2 (Sketch)): The termination and validity properties of uniform consensus and quiescence are similarly satisfied as in the case of the algorithms in Sect. 4. The interesting part is the uniform agreement property—thereby the reintegration of recovered processes in the crash-stop consensus run—if no decision occurred so far. Otherwise, the decision value is already saved on stable storage, or a recovered process is informed by any currently up process. In order to guarantee uniform agreement the important information of the last estimate and its adoption time is either restored with the last message, which is loaded from stable storage, or never occurred so far in the time that a recovered process participated previously in the crash-stop consensus run. This fact is important for the correctness proof of the algorithm. ■

5.3 Transformation Complexity

The additional number of messages and rounds is roughly the analysis in Sect. 4.4. One additional advantage of stable storage is that if all recovering processes already stored the decision on stable storage, no message at all needs to be sent.

Algorithm 1 Emulator with $\Diamond\mathcal{P}$, stable storage, and “correct majority” assumption

Implements:
Uniform Consensus (**propose**, **decide**)

Uses:
CS-consensus (**CS-propose**, **CS-decide**, **CS-suspect**, **CS-send**,
CS-receive, **CS-single-send**, **CS-stop-retransmit**)
Stubborn Links (**send**, **receive**, **single-send**, **stop-retransmit**)
Eventually Perfect Failure Detector (**suspect**)
Stable Storage (**save**, **load**)

Assumption: majority of correct processes.

The same algorithm runs on every process $p \in \{1, \ldots, n\}$.

```
upon init do
    $Suspected_p \leftarrow \emptyset$
    $proposal_p \leftarrow decisionvalue_p \leftarrow received_p \leftarrow sent_p \leftarrow \bot$

upon propose value $v$ do
    save $v$ at PROPOSAL
    CS-propose $v$

upon CS-send message $m$ to process $q$ do
    if $sent_p \prec m$ then
        save $m, q$ at SENT
        $sent_p \leftarrow m$
    send $\langle$CS-CONSENSUS$, m\rangle$ to $q$

upon CS-single-send message $m$ to process $q$ do
    single-send $\langle$CS-CONSENSUS$, m\rangle$ to $q$

upon receive $\langle$CS-CONSENSUS$, m\rangle$ from process $q$ do
    if $decisionvalue_p = \bot$ then
        if $received_p \prec m$ then
            save $m, q$ at RECEIVED
            $received_p \leftarrow m$
        CS-receive $m$ from $q$
    else
        single-send $\langle$DECIDE$, decisionvalue_p\rangle$ to $q$

upon CS-decide value $v$ do
    $decisionvalue_p \leftarrow v$
    send $\langle$DECIDE$, decisionvalue_p\rangle$ to all
```

The algorithm continues on page 14.

Part 2 of algorithm 1 Emulator with $\Diamond\mathcal{P}$, stable storage, and “correct majority” assumption

```
upon receive $\langle \text{DECIDE}, decisionvalue_q \rangle$ do
    $decisionvalue_p \leftarrow decisionvalue_q$
    save $decisionvalue_p$ at DECISION
    decide $decisionvalue_p$
    stop-retransmit

upon suspect set of processes $Q$ do                              ▷ Algorithm uses $\Diamond\mathcal{P}$
    $Suspected_p \leftarrow Q$
    CS-suspect $Suspected_p$

upon recovery do
    $Suspected_p \leftarrow \emptyset$
    load $decisionvalue_p$ at DECISION
    if $decisionvalue_p \neq \perp$ then
        decide $decisionvalue_p$
        stop-retransmit
    else
        load $proposal_p$ at PROPOSAL
        if $proposal_p \neq \perp$ then
            load $received_p, from_p$ at RECEIVED
            load $sent_p, to_p$ at SENT
            CS-propose $proposal_p$
            if $received_p \neq \perp$ then
                CS-receive $received_p$ from $from_p$
            if $sent_p \neq \perp$ then
                send $\langle \text{CS-CONSENSUS}, sent_p \rangle$ to $to_p$
            else
                send $\langle \text{CS-CONSENSUS}, \perp \rangle$ to all
```

References

1. M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Workshop on Distributed Algorithms*, pages 126–140, 1997.
2. M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash recovery model. *Distributed Computing*, 13(2):99–125, Apr. 2000.
3. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
4. M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.
5. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
6. R. Guerraoui, R. C. Oliveira, and A. Schiper. Stubborn communication channels. Technical report, Département d'Informatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland, December 1996.
7. R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag, Berlin, Germany, 2006.
8. M. Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, pages 280–286, West Lafayette, Indiana, Oct. 1998. IEEE Computer Society Press.
9. L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
10. R. C. Oliveira. *Solving consensus: from fair-lossy channels to crash-recovery of processes.* PhD thesis, Ecole Polytechnique Federale de Lausanne, 2000.
11. R. C. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report 97-239, Département d'Informatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland, August 1997.

A Modular Algorithm based on $\mathcal{P}$

Algorithm 2 solves uniform consensus in the crash-recovery model with the help of a perfect failure detector and the assumption of the presence of at least one always up process. The proof of Theorem 1 follows. First, some lemmata are proven to increase the readability of the theorem's proof.

Lemma 3 (Validity): If a process decides value v, then v was proposed by some process.

Proof: Since only up—and not recovered—processes participate in the crash-stop consensus algorithm, the lemma follows by the validity property of the used crash-stop consensus algorithm. ■

Lemma 4 (Uniform Agreement): Processes do not decide different values.

Proof: Consider the three possible cases of process failure times:

i) Never: Since these processes run a crash-stop consensus algorithm and no one of them crashes, the claim follows by the uniform agreement property of the used crash-stop consensus algorithm and by the stubbornness property of the links.

ii) After their crash-stop consensus algorithm terminated: Analogous to i). After their recovery only the decision value of the crash-stop consensus algorithm is disseminated.

iii) Before: After their recovery only the decision value of the crash-stop consensus algorithm is disseminated.

■

Lemma 5 (Termination): Every correct process eventually decides.

Proof: All always up processes terminate because of the termination property of the crash-stop consensus algorithm, and it is assumed that at least one always up process is present in the system. After the termination of the crash-stop consensus algorithm the decision value is disseminated to all other processes, and all "I recovered" messages are answered with the instruction to decide. Thus, by the stubbornness property of the communication links, all correct processes eventually decide. ■

Lemma 6 (Quiescence): The algorithm is quiescent, i.e., eventually it stops sending messages, if no process is unstable.

Proof: Since all correct processes eventually decide—confer lemma 5—they execute line 18 of the algorithm and stop sending any message. If an unstable process is present in the system and recovers, it starts sending "I recovered" messages in line 25 of the algorithm. The responses to these messages by the other processes are kept to a minimum, because they are not sent stubbornly—confer line 28. ■

Proof (Theorem 1): By lemmata 3, 4, and 5 all consensus properties are satisfied and by lemma 6 quiescence is also satisfied. Thus, the theorem holds. ■

B Crash-Stop Consensus Algorithm bases on $\Diamond\mathcal{P}$

Algorithm 3, which originally appeared elsewhere [3], uses the rotating coordinator paradigm for consensus with failure detectors, in order to solve uniform consensus in the crash-stop model with the help of an eventually perfect failure detector and the assumption of the presence of a majority of correct processes.

Algorithm 2 Emulator with $\mathcal{P}$ and “at least one always up” assumption

```
Implements:
    Uniform Consensus (propose, decide)

Uses:
    CS-consensus (CS-propose, CS-decide, CS-suspect, CS-send
        CS-receive, CS-single-send, CS-stop-retransmit)
    Stubborn Links (send, receive, single-send, stop-retransmit)
    Perfect Failure Detector (suspect)

Assumption: at least one always up process.

The same algorithm runs on every process $p \in \{1, \dots, n\}$.
upon init do
    $Suspected_p \leftarrow \emptyset$
    $decisionvalue_p \leftarrow \perp$

upon propose value $v$ do
    CS-propose $v$

upon CS-send message $m$ to process $q$ do
    send $\langle$CS-CONSENSUS, $m\rangle$ to $q$

upon CS-single-send message $m$ to process $q$ do
    single-send $\langle$CS-CONSENSUS, $m\rangle$ to $q$

upon receive $\langle$CS-CONSENSUS, $m\rangle$ from process $q$ do
    CS-receive $m$ from $q$

upon CS-decide value $v$ do
    $decisionvalue_p \leftarrow v$
    send $\langle$DECIDE, $decisionvalue_p\rangle$ to all

upon receive $\langle$DECIDE, $decisionvalue_q\rangle$ do
    $decisionvalue_p \leftarrow decisionvalue_q$
    decide $decisionvalue_p$
    stop-retransmit

upon suspect set of processes $Q$ do                ▷ Algorithm uses $\mathcal{P}$
    $Suspected_p \leftarrow Suspected_p \cup Q$
    CS-suspect $Suspected_p$

upon recovery do
    $Suspected_p \leftarrow \emptyset$
    $decisionvalue_p \leftarrow \perp$
    send $\langle$RECOVERED$\rangle$ to all

upon receive $\langle$RECOVERED$\rangle$ from process $q$ do
    if $decisionvalue_p \neq \perp$ then
        single-send $\langle$DECIDE, $decisionvalue_p\rangle$ to $q$
    $Suspected_p \leftarrow Suspected_p \cup \{q\}$
    CS-suspect $Suspected_p$
```

Algorithm 3 Crash-stop consensus algorithm that uses $\Diamond\mathcal{P}$

Implements:
Uniform Consensus (**propose**, **decide**)

Uses:
Stubborn Links (**send**, **receive**, **single-send**, **stop-retransmit**)
Eventually Perfect Failure Detector (**suspect**)

Assumption: majority of correct processes.

The same algorithm runs on every process $p \in \{1, \ldots, n\}$.

```
upon init do
    $Suspected_p \leftarrow \emptyset$
    $decisionvalue_p \leftarrow estimate_p \leftarrow \perp$
    $round_p \leftarrow leader_p \leftarrow \infty$

upon propose value $v$ do
    if $estimate_p = \perp$ then
        $estimate_p \leftarrow v$
        $adopted_p \leftarrow 0$
        $round_p \leftarrow 1$
        $leader_p \leftarrow \big((round_p - 1) \bmod n\big) + 1$
        $Estimates_p \leftarrow Acks_p \leftarrow \emptyset$
        send $\langle round_p, \text{WAKEUP} \rangle$ to $leader_p$

upon $p = leader_p$ and $|Estimates_p| = 0$ do
    send $\langle round_p, \text{NEWROUND} \rangle$ to all

upon receive $\langle round_p, \text{NEWROUND} \rangle$ do
    send $\langle round_p, \text{ESTIMATE}, estimate_p, adopted_p \rangle$ to $leader_p$

upon receive $\langle round_p, \text{ESTIMATE}, estimate_q, adopted_q \rangle$ from process $q$ do
    $Estimates_p \leftarrow Estimates_p \cup \{(q, estimate_q, adopted_q)\}$

upon $p = leader_p$ and $|Estimates_p| \geq \lceil \frac{n+1}{2} \rceil$ do
    $\exists\, (q, estimate_q, adopted_q) \in Estimates_p\colon$
      $\forall\, (q', e_q, a_q) \in Estimates_p\colon a_q \leq adopted_q$
    $estimate_p \leftarrow estimate_q$
    $adopted_p \leftarrow round_p$
    send $\langle round_p, \text{ADOPT}, estimate_p \rangle$ to all

upon receive $\langle round_p, \text{ADOPT}, estimate_q \rangle$ do
    $estimate_p \leftarrow estimate_q$
    $adopted_p \leftarrow round_p$
    send $\langle round_p, \text{ACK} \rangle$ to $leader_p$
```

The algorithm continues on page 19.

Part 2 of algorithm 3 Crash-stop consensus algorithm that uses $\Diamond\mathcal{P}$

```
upon receive $\langle round_p, \text{ACK}\rangle$ from process $q$ do
    $Acks_p \leftarrow Acks_p \cup \{q\}$

upon $p = leader_p$ and $|Acks_p| \geq \lceil \frac{n+1}{2} \rceil$ do
    send $\langle \text{DECIDE}, estimate_p \rangle$ to all

upon receive $\langle \text{DECIDE}, estimate_q \rangle$ and $decisionvalue_p = \bot$ do
    $decisionvalue_p \leftarrow estimate_q$
    decide $decisionvalue_p$
    $leader_p \leftarrow round_p \leftarrow \infty$                                   ▷ Stops further events
    stop-retransmit

upon receive message $m$ from process $q$ and $decisionvalue_p \neq \bot$ do
    single-send $\langle \text{DECIDE}, decisionvalue_p \rangle$ to $q$

upon suspect set of processes $Q$ do                                     ▷ Algorithm uses $\Diamond\mathcal{P}$
    $Suspected_p \leftarrow Q$

upon $leader_p \in Suspected_p$ do
    $round_p \leftarrow round_p + 1$
    $leader_p \leftarrow \big((round_p - 1) \bmod n\big) + 1$
    $Estimates_p \leftarrow Acks_p \leftarrow \emptyset$
    send $\langle round_p, \text{WAKEUP} \rangle$ to $leader_p$

upon receive $\langle round_q, \ldots \rangle$ and $round_q > round_p$ do
    $round_p \leftarrow round_q$
    $leader_p \leftarrow \big((round_p - 1) \bmod n\big) + 1$
    $Estimates_p \leftarrow Acks_p \leftarrow \emptyset$
    simulate receive $\langle round_q, \ldots \rangle$
```