Modular Consensus Algorithms for the Crash-Recovery Model

Felix C. Freiling, Christian Lambertz, and Mila Majster-Cederbaum
Department of Computer Science, University of Mannheim, Germany
E-mail: {freiling, lambertz, mcb}@informatik.uni-mannheim.de

***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 consensus algorithms for the crash-recovery model for different types of assumptions.**

***Index Terms*—asynchronous systems, consensus, fault tolerance, process crash and recovery, stable storage**

I. Introduction

A. 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 interesting model, especially if it is paired with the *asynchronous message-passing system* model of distributed computations. In this model, processes communicate by sending messages, 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* [7].

Despite its theoretical interest, 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). Note that the processes in class (1) and (2) are called *correct*, and those in class (3) and (4) *incorrect*. 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.

The additional expressiveness of the crash-recovery model makes it relatively hard to design algorithms for this model. In this paper, we aim to develop algorithms for this model by re-using algorithms from the crash-stop model, i.e., we target at *modular* solutions.

B. Consensus in the Crash-Recovery Model

We choose the problem of *distributed consensus* as benchmark problem to study the modular derivation of algorithms from crash-stop to 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 be solved deterministically in asynchronous systems [7], but becomes solvable using extensions of the model. In this paper, we use the *failure detector* abstraction [4] 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 leads 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* [9]. Briefly spoken, such channels infinitely often re-send a message as long as the sender does not crash. Therefore, a message sent over a stubborn channel will eventually be received at its destination as long as the receiver eventually recovers.

C. Related Work

A seminal paper on consensus in the crash-recovery model was published by Aguilera, Chen, and Toueg [1]. 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 algorithm is constructed which assumes the presence of a majority of correct processes.

Oliveira, Guerraoui, and Schiper [14] 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 [10] developed a complex voting based consensus algorithm that uses stable storage. Both papers [10], [14] use a failure detector definition that was later shown [1] 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 [11] also solves consensus in the setting of this paper. While re-using algorithmical ideas from the area of crash-stop algorithms, Paxos as well as all previously mentioned algorithms were built from scratch.

Modular derivations of fault-tolerant consensus algorithms have been extensively studied before [2], [13], [6], [3], [12], [5] but in different contexts than our work. Neiger and Toueg [13], Delporte-Gallet et al. [6], and Bazzi and Neiger [3] assume synchronous systems and no failure detectors. Asynchronous systems are considered by Basu, Charron-Bost, and Toueg [2] but in the context of link failures and also without failure detectors. Mittal et al. [12] consider the problem of termination detection and derive algorithms for the crash-stop model from algorithms that assume no failures.

D. Contributions

Similarly to Aguilera, Chen, and Toueg [1] we investigate the solvability of consensus in the crash-recovery model under varying assumptions. 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. We are able to close many previously open cases and present a complete map of consensus solvability in the crash-recovery model under varying assumptions.

Table I gives an overview over the cases we study in this paper. The table is structured 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 "$\checkmark$". 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.

As mentioned before, the case of no stable storage, $\Diamond\mathcal{P}$, and more always up than incorrect processes was proven to be the weakest for consensus [1], 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. III-A that at least one always up process is necessary and sufficient. The sufficiency part is proved in Sect. III-B. Therefore, consensus is also solvable under stronger process assumptions. Then we weaken $\mathcal{P}$ to $\Diamond\mathcal{P}$ and present a modular algorithm for the always up majority of processes case in Sect. III-C. This completes the discussion for the case with no stable storage.

We then turn in Sect. IV-A 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 $\Diamond\mathcal{P}$ and one always up process. The second one arises in the case where $\mathcal{P}$ is available and one eventually up process is present. We then give an algorithm for the remaining case in Sect. IV-B: It uses $\Diamond\mathcal{P}$, a majority of always up or eventually up processes, and some minimal insight about the used crash-stop consensus

assumptions	**no stable storage**		**stable storage**	
	$\Diamond\mathcal{P}$	$\mathcal{P}$	$\Diamond\mathcal{P}$	$\mathcal{P}$
one correct	×	×	×	× Sect. IV-A
correct majority	×	× Sect. III-A	✓ Sect. IV-B, [1]	✓
one always up	×	✓ Sect. III-B	× Sect. IV-A	✓ Sect. III-B
correct majority & one always up	× [1]	✓	✓	✓
more always up than incorrect	✓ [1]	✓	✓ [1]	✓
always up majority	✓ Sect. III-C	✓	✓ Sect. III-C	✓

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

algorithm which needs to be saved on stable storage. But, with this insight the algorithm is not completely modular, and thus we call it semi-modular.

Note that we use failure detectors with strong accuracy properties instead of detectors with weak accuracy properties, although we are aware that consensus is solvable with weaker failure detectors in the crash-recovery model [1]. Due to pedagogical reasons, we prefer to present our approach with the stronger assumptions because weakening them does not add any insight to understand our approach. Additionally, this weakening of the accuracy property of our failure detectors is possible without any further change in our emulation algorithms.

Since we assume that the consensus problem (with its termination, uniform agreement, and validity properties), the crash-recovery model, and the asynchronous system are well-known, we omit their formal definition here. These, together with pseudo code of our algorithms and correctness proofs, can be found elsewhere [8]. Here, we only include the definition of the two failure detectors. A *perfect failure detector* ($\mathcal{P}$) satisfies: *Strong Completeness:* Every incorrect process is suspected infinitely often by every correct process, *Eventually Up Completeness:* Eventually, every correct process is not suspected any longer by every correct process, and *Strong Accuracy:* No process is suspected before it crashes. An *eventually perfect failure detector* ($\Diamond\mathcal{P}$) satisfies *Strong Completeness*, *Eventually Up Completeness*, and *Eventually Strong Accuracy:* Correct processes are only finitely often suspected.

II. The Emulation Technique and its Limitations

A. 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}. 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 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}.

B. Limitations of Modular Solutions

Aguilera, Chen, and Toueg [1] 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 algorithm 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 [1] present such a (non-modular) algorithm.

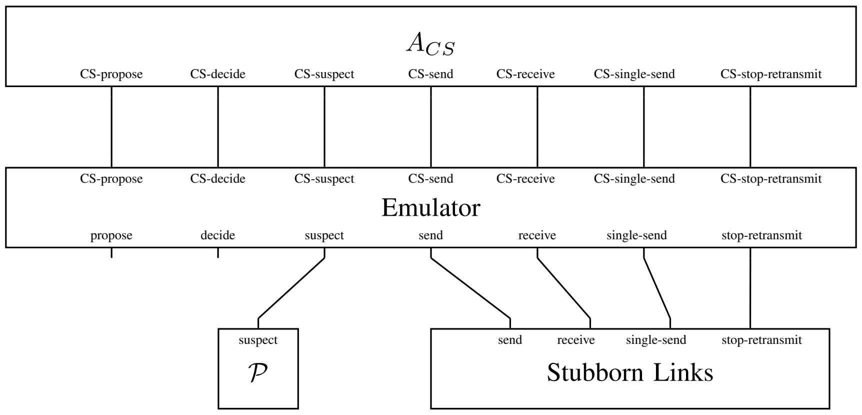

FIG. 1: An emulator algorithm and its connection to the A_{CS} interface.

III. Modular 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.

A. 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. III-B, 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 [4]). 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 value due to crashes [1].

B. Modular Algorithm based on $\mathcal{P}$

Assuming at least one always up process, we now present a modular 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. Furthermore, all messages of A_{CS} are tagged with a special identifier in order to prevent the reception of any non-A_{CS} message by the A_{CS} module. 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.

C. 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 previous one. The main difference is the handling of the false suspicions made by $\diamond\mathcal{P}$. In order to separate the possibly falsely suspected processes and the recovered ones, two sets are defined: *Suspected*$_p$ and *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 *Suspected*$_p$ is updated with any change in the output 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.

Interestingly, the always up majority assumption is only needed until A_{CS} terminates. After the decision of A_{CS} happened, the emulator needs only at least one always up process in order to disseminate the decision value.

D. *Transformation Complexity*

Both presented algorithms do not increase the complexity of the used crash-stop consensus algorithm much. Let n denote the number of processes and n_{correct} the number of correct processes. Assume that A_{CS} needs m_{cs} messages and r_{cs} rounds to find a decision. If no recoveries occur, at most $n_{\text{correct}} \times n$ additional decision messages—every correct process broadcasts the decision—are sent and only one more round is needed. Since typically $m_{cs} > n_{\text{correct}} \times n$, we obtain the same bounds.

With every recovery $n + n_{\text{correct}}$ 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 n 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.

IV. Modular Consensus Algorithms Using Stable Storage

Aguilera, Chen, and Toueg [1] 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 if less always up than incorrect processes are present. 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. IV-B 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. IV-A.

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. III-B). We prove that this is a minimal requirement also for systems with stable storage in the following section.

A. *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. Consensus is not solvable if $\Diamond\mathcal{P}$ and stable storage are accessible by the processes and if only one always up process is assumed to be present in the system, i.e., no majority of correct processes is present.

To see this, 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 v in an analogues way, because its failure detector suspects p_2. But, this second decision violates the uniform agreement property of consensus.

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.

To see this, 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.

B. *Semi-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 message 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 have to use a concrete crash-stop consensus algorithm.

Imagine the following crash-stop consensus algorithm inspired by the well-known Chandra/Toueg rotating coordinator consensus algorithm [4]: 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 [4].

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 the above mentioned algorithm is used as A_{CS}, the most essential state is the successful adoption of a future decision value, i.e., when a process received an adopt message from the current round leader and sends back an acknowledgment message. This situation means that the process agrees to the decision of a certain value. Thus, this state information must be remembered as the most important last sent/received message.

The algorithm of Sect. III-C can be extended to save the last message information on stable storage. The important difference is the new assumption that only a majority of correct processes is needed and not a majority of always up processes as in the case of the algorithm of Sect. III-C.

If 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 message 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, the emulator broadcasts an empty message 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. Thereby, the empty message must be the lowest message in the message order of the used crash-stop consensus algorithm.

C. Transformation Complexity

The additional number of messages and rounds is roughly the same as in the analysis in Sect. III-D. 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.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[2] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings in the 10th International Workshop on Distributed Algorithms*, pages 105–122, 1996.

[3] R. A. Bazzi and G. Neiger. Simulating crash failures with many faulty processors (extended abstract). In *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 166–184, 1992.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[5] C. Delporte-Gallet, H. Fauconnier, F. C. Freiling, L. D. Penso, and A. Tielmann. From crash-stop to permanent omission: Automatic transformation and weakest failure detectors. In *Proc. DISC*, 2007.

[6] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and B. Pochon. The perfectly-synchronised round-based model of distributed computing. *Information & Computation*, 2007.

[7] 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.

[8] F. C. Freiling, C. Lambertz, and M. Majster-Cederbaum. Easy consensus algorithms for the crash-recovery model. Technical Report TR-2008-002, Department of Computer Science, University of Mannheim, Germany, 2008.

[9] R. Guerraoui, R. C. Oliveira, and A. Schiper. Stubborn communication channels. Technical report, Département d'Informatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland, December 1996.

[10] M. Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 280–286, 1998.

[11] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[12] N. Mittal, F. C. Freiling, S. Venkatesan, and L. D. Penso. Efficient reduction for wait-free termination detection in a crash-prone distributed system. In *Proc. 19th Conference on Distributed Computing*, pages 93–107, 2005.

[13] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[14] 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, 1997.