Distrib. Comput. (2000) 13: 99–125

DISTRIBUTED COMPUTING

Failure detection and consensus in the crash-recovery model

Marcos Kawazoe Aguilera[1], **Wei Chen**[2], **Sam Toueg**[1]

[1] Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, USA (e-mail: {aguilera,sam}@cs.cornell.edu)
[2] Oracle Corporation, One Oracle Drive, Nashua, NH 03062, USA (e-mail: weichen@us.oracle.com)

Received: May 1998 / Accepted: November 1999

Summary. We study the problems of failure detection and consensus in asynchronous systems in which processes may crash and recover, and links may lose messages. We first propose new failure detectors that are particularly suitable to the crash-recovery model. We next determine under what conditions stable storage is necessary to solve consensus in this model. Using the new failure detectors, we give two consensus algorithms that match these conditions: one requires stable storage and the other does not. Both algorithms tolerate link failures and are particularly efficient in the runs that are most likely in practice – those with no failures or failure detector mistakes. In such runs, consensus is achieved within 3δ time and with $4n$ messages, where δ is the maximum message delay and n is the number of processes in the system.

Key words: Fault tolerance – Failure detection – Consensus – Process crash – Process recovery – Asynchronous systems – Stable storage

1 Introduction

The problem of solving consensus in asynchronous systems with unreliable failure detectors (i.e., failure detectors that make mistakes) was first investigated in [3, 2]. But these works only considered systems where process crashes are *permanent* and links are reliable (i.e., they do not lose messages). In real systems, however, processes may *recover* after crashing and links may lose messages. In this paper, we focus on solving consensus with failure detectors in such systems, a problem that was first considered in [4, 10, 7] (a brief comparison with these works is in Sect. 1.3).

Solving consensus in a system where process may recover after crashing raises two new problems; one regards the need for stable storage and the other is about the failure detection requirements:

- *Stable Storage:* When a process crashes, it loses all its local state. One way to deal with this problem is to assume that parts of the local state are recorded into stable storage, and can be restored after each recovery. However, stable storage operations are slow and expensive, and must be avoided as much as possible. Is stable storage always necessary when solving consensus? If not, under which condition(s) can it be completely avoided?
- *Failure Detection:* In the crash-recovery model, a process may keep on crashing and recovering indefinitely (such a process is called *unstable*). How should a failure detector view unstable processes? Note that an unstable process may be as useless to an application as one that permanently crashes (and in fact it could be even more disruptive). For example, an unstable process can be up just long enough to be considered operational by the failure detector, and then crash before "helping" the application, and this could go on repeatedly. Thus, it is natural to require that a failure detector satisfies the following *completeness* property: Eventually every unstable process is permanently suspected.[1] But implementing such a failure detector is inherently problematic *even in a perfectly synchronous system*. Intuitively, this is because, at any given point in time, no implementation can predict the future behavior of a process p that has crashed in the past but is currently "up". Will p continue to repeatedly crash and recover? Or will it stop crashing?

In summary, our goal here is to solve consensus in the crash-recovery model (with lossy links). As a crucial part of this problem, we first need to find reasonable failure detectors that can be used for this task. We also need to determine if and when stable storage is necessary.

1.1 Failure detectors for the crash-recovery model

We first focus on the problem of failure detection in the crash-recovery model. Previous solutions require unstable processes

Research partially supported by NSF grant CCR-9402896 and CCR-9711403, by ARPA/ONR grant N00014-96-1-1014, and by an Olin Fellowship.

[1] In fact, this property is assumed in [10, 7].

to be eventually suspected forever [10,7].[2] We first prove that this requirement has a serious drawback: it forces failure detector implementations to have undesirable behaviors even in perfectly synchronous systems. More precisely, consider a synchronous round-based system with no message losses,[3] where up to n_u processes may be unstable. In this system, *every* implementation of a failure detector with the above requirement has runs with the following undesirable behavior: there is a round after which (a) *all* processes are permanently up, but (b) the failure detector incorrectly suspects n_u of them forever (see Theorem 1). Note that these permanent mistakes are *not* due to the usual causes, namely, slow processes or message delays. Instead, they are entirely due to the requirement on unstable processes (which involves predicting the future).

To avoid the above problem, we propose *a new type of failure detector* that is well-suited to the crash-recovery model. This failure detector does not output lists of processes suspected to be crashed or unstable. Instead, it outputs a list of processes deemed to be currently up, with an associated *epoch number* for each such process. If a process is on this list we say it is *trusted*.

The epoch number of a process is a rough estimate of the number of times it crashed and recovered in the past. We distinguish two types of processes: *bad* ones are those that are unstable or crash permanently, and *good* ones are those that never crash or eventually remain up. We first propose a simple failure detector, denoted $\diamond\mathcal{S}_e$, with the following two properties. Roughly speaking (precise definitions are in Sect. 3):

- *Completeness:* For every bad process b, at every good process there is a time after which either b is never trusted or the epoch number of b keeps on increasing.
- *Accuracy:* Some good process is eventually trusted forever by all good processes, and its epoch number stops changing.

Note that the completeness property of $\diamond\mathcal{S}_e$ does not require predicting the future (to determine if a process is unstable), and so it does not force implementations to have anomalous behaviors. To illustrate this, in Appendix B we give an implementation of $\diamond\mathcal{S}_e$ for some models of partial synchrony: this implementation ensures that if all processes are eventually up forever they will be eventually trusted forever.

Failure detector $\diamond\mathcal{S}_e$, however, does not put *any* restriction on how the bad processes view the system. In particular, the accuracy property allows unstable processes to repeatedly "suspect" *all* processes.[4] This is problematic because, in contrast to processes that permanently crash, unstable processes may continue to take steps, and so their incorrect suspicions may prevent the progress of some algorithms. For example, in the rotating coordinator consensus algorithms of [3,4,7] if a process kept suspecting all processes then consensus would never be reached.

From the above it is clear that sometimes it is better to have a failure detector with:

- *Strong Accuracy:* Some good process is eventually trusted forever by all good *and unstable* processes, and its epoch number stops changing.

Such a failure detector is denoted $\diamond\mathcal{S}_u$. In this paper, we show how to transform any $\diamond\mathcal{S}_e$ into $\diamond\mathcal{S}_u$ in an asynchronous system provided that a majority of processes are good.

1.2 On the necessity of stable storage in the crash-recovery model

Can consensus be solved in the crash-recovery model *without stable storage*, and if so, how? To answer this question, assume that during each execution of consensus, at least n_a processes are guaranteed to remain up, and at most n_b processes are bad.

Clearly, if $n_a < 1$ then consensus cannot be solved without stable storage: it is possible that *all* processes crash and recover during execution, and the entire state of the system (including previous proposals and possible decisions) can be lost forever. On the other hand, if $n_a > n/2$, i.e., a majority of processes are guaranteed to remain up, then solving consensus without stable storage is easy: If a process crashes we exclude it from participating in the algorithm even if it recovers (except that we allow it to receive the decision value). This essentially reduces the problem to the case where process crashes are permanent and a majority of processes do not crash (and then an algorithm such as the one in [3] can be used).

Is it possible to solve consensus without stable storage if $1 \leq n_a \leq n/2$? We show that:

- If $n_a \leq n_b$ then consensus *cannot be solved without stable storage* even using $\diamond\mathcal{P}$ (the *eventually perfect failure detector* defined in Sect. 5).
- If $n_a > n_b$ then consensus *can be solved without stable storage* using $\diamond\mathcal{S}_e$ (which is weaker than $\diamond\mathcal{P}$).

This last result is somewhat surprising because with $n_a > n_b$, *a majority of processes may crash and completely lose their state* (including the consensus values they may have previously proposed and/or decided). To illustrate this with a concrete example, suppose $n = 10$, $n_a = 3$ and $n_b = 2$. In this case, up to 7 processes — more than half of the processes — may crash and lose their state, and yet consensus is solvable with a failure detector that is weaker than $\diamond\mathcal{P}$. *Prima facie*, this seems to contradict the fact that if a majority of processes may crash then consensus cannot be solved even with $\diamond\mathcal{P}$ [3]. There is no contradiction, however, since [3] assumes that all process crashes are permanent, while in our case some of the processes that crash do recover: even though they completely lost their state, they can still provide some help.

What if stable storage *is* available? In this case, we show that consensus can be solved with $\diamond\mathcal{S}_u$, provided that a majority of processes are good (this majority requirement is weaker than $n_a > n_b$). Note that if the good processes are not a majority, then consensus cannot be solved even with $\diamond\mathcal{P}$ [3].

In addition to crashes and recoveries, the two consensus algorithms that we give (with and without stable storage) also tolerate *message losses*, provided that links are fair lossy, i.e., if p sends messages to a good process q infinitely often, then q receives messages from p infinitely often.

[2] In [4], crash-recovery is regarded as a special case of omission failures, and the algorithm is not designed to handle unstable processes that can send and receive messages to and from good processes.

[3] In such a system, processes execute in synchronized rounds, and all messages are received in the round they are sent.

[4] Intuitively, this is because an unstable process may fail to receive "I am alive" messages sent by other processes since all messages that "arrive" at a process while it is down are lost.

1.3 Related work

The problem of solving consensus with failure detectors in systems where processes may recover from crashes was first addressed in [4] (with crash-recovery as a form of omission failures) and more recently studied in [10, 7].

In [4, 10, 7], the question of whether stable storage is always necessary is not addressed, and all the algorithms use stable storage: in [4, 10], the entire state of the algorithm is recorded into stable storage at every state transition; in [7], only a small part of the state is recorded, and writing to stable storage is done at most once per round. In this paper, we determine when stable storage is necessary, and give two matching consensus algorithms — with and without stable storage. In the one that uses stable storage, only a small part of the state is recorded and this occurs twice per round.

The algorithms in [10, 7] use failure detectors that require that unstable processes be eventually suspected forever. The algorithm in [4] is not designed to deal with unstable processes which may intermittently communicate with good ones.

1.4 Summary of results

We study the problems of failure detection and consensus in asynchronous systems with process crashes and recoveries, and lossy links.

1. We show that the failure detectors that have been previously proposed for the crash-recovery model with unstable processes have inherent drawbacks: Their completeness requirement force implementations to have anomalous behaviors even in synchronous systems.
2. We propose new failure detectors that avoid the above drawbacks.
3. We determine under what conditions stable storage is necessary to solve consensus in the crash-recovery model.
4. We give two consensus algorithms that match these conditions, one uses stable storage and the other does not. Both algorithms tolerate message losses, and are particularly efficient in the runs that are most likely in practice — those with no failures or failure detector mistakes. In such runs, consensus is achieved within 3δ time and with $4n$ messages, where δ is the maximum message delay and n is the number of processes in the system.

1.5 Roadmap

The paper is organized as follows. Our model is given in Sect. 2. In Sect. 3 we show that existing failure detectors for the crash-recovery model have limitations, and then introduce our new failure detectors, namely $\diamond\mathcal{S}_e$ and $\diamond\mathcal{S}_u$. We define the Consensus problem in Sect. 4. In Sect. 5, we determine under what conditions consensus requires stable storage. We then give two matching consensus algorithms: one does not require stable storage (Sect. 6), and the other uses stable storage (Sect. 7). In Sect. 8, we briefly consider the performance of these algorithms. The issue of repeated consensus is discussed in Sect. 9. In Sect. 10, we show how to transform $\diamond\mathcal{S}_e$ into $\diamond\mathcal{S}_u$.

2 Model

We consider asynchronous message-passing distributed systems in which there are no timing assumptions. In particular, we make no assumptions on the time it takes to deliver a message, or on relative process speeds. We assume that every process is connected with every other process through a communication link. Links can fail by intermittently dropping messages. A process can fail by crashing and it may subsequently recover. When a process crashes it loses all of its state. However, it may use local stable storage to save (and later retrieve) parts of its state.

We assume the existence of a discrete global clock — this is merely a fictional device to simplify the presentation and processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

2.1 Processes and process failures

The system consists of a set of n processes, $\Pi = \{1, 2, \ldots, n\}$. Processes can crash and may subsequently recover. A *failure pattern* F is a function from $\mathcal{T}$ to 2^{Π}. Intuitively, $F(t)$ denotes the set of processes that are not functioning at time t. We say process p is *up at time* t *(in* F*)* if $p \notin F(t)$ and p is *down at time* t *(in* F*)* if $p \in F(t)$. We say that p *crashes at time* t if p is up at time $t-1$ and p is down at time t.[5] We say that p *recovers* at time $t \geq 1$ if p is down at time $t-1$ and p is up at time t. A process p can be classified (according to F) as *always-up*, *eventually-up*, *eventually-down* and *unstable* as follows:

Always-up: Process p never crashes.
Eventually-up: Process p crashes at least once, but there is a time after which p is permanently up.
Eventually-down: There is a time after which process p is permanently down.
Unstable: Process p crashes and recovers infinitely many times.

A process is *good (in* F*)* if it is either always-up or eventually-up. A process is *bad (in* F*)* if it is not good (it is either eventually-down or unstable). We denote by $good(F)$, $bad(F)$ and $unstable(F)$ the set of good, bad and unstable processes in F, respectively. Henceforth, we consider only failure patterns with at least one good process.

2.2 Failure detectors

Each process has access to a local failure detector module that provides (possibly incorrect) information about the failure pattern that occurs in an execution. A process can query its local failure detector module at any time. A *failure detector history* H *with range* $\mathcal{R}$ is a function from $\Pi \times \mathcal{T}$ to $\mathcal{R}$. $H(p, t)$ is the output value of the failure detector module of process p at time t. A *failure detector* $\mathcal{D}$ is a function that maps each failure pattern F to a set of failure detector histories with range $\mathcal{R}_{\mathcal{D}}$ (where $\mathcal{R}_{\mathcal{D}}$ denotes the range of the failure detector output of $\mathcal{D}$). $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for the failure pattern F.

[5] We say that p crashes at time $t = 0$ if p is down at time 0.

2.3 Stable storage

When a process crashes, it loses all its volatile state, but we assume that when it recovers, it knows that it is recovering from a crash. Moreover, a process may use a stable storage device to store and retrieve a set of variables. These two stable storage operations cannot be executed atomically with certain other actions. For example, a process cannot store a variable in stable storage and then send a message or issue an external output, in a single atomic step. The actions that a process can execute in an atomic step are detailed in the next section.

2.4 Runs of algorithms

An algorithm A is a collection of n deterministic automata, one for each process in the system. Computation proceeds in atomic *steps* of A. There are two types of steps: a *crash step* and a *normal step*. In a *crash step*, the state of a process is changed to a specially designated state called the *crash state* (thus the process "loses its state"). In a *normal step*, a process:

1. First executes *one* of the following actions, according to its state: (a) store a set of variables into local stable storage, (b) retrieve a set of variables from local stable storage, (c) send a message to some process, or (d) issue an external output.[6]
2. Then it attempts to execute *each one* of the following actions: (a) receive a message from a process, (b) get an external input, and (c) query its failure detector;
3. Finally, it changes state.

An *initial configuration of algorithm A* consists of the initial state of the automaton for each process. A *run of algorithm A using failure detector $\mathcal{D}$* is a tuple $R = (F, H_{\mathcal{D}}, I, S, T)$ where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is a history of failure detector $\mathcal{D}$ for failure pattern F, I is an initial configuration of A, S is an infinite sequence of steps of A, and T is an infinite list of non-decreasing time values indicating when each step in S occurs.

A run must satisfy the following properties: (1) a process takes at most one step at each time t; (2) a process takes a normal step at time t only if it is up at t; (3) a process takes a crash step at time t if and only if it crashes at t; (4) a good process takes an infinite number of normal steps; (5) if a process p takes a step at time t and queries its failure detector, then it obtains $H_{\mathcal{D}}(p, t)$ as a response; (6) when a process retrieves a variable from stable storage, it obtains the last value that it stored for that variable (or $\perp$ if it never stored the variable).

Note that if a process p recovers from a crash, its first step is from the special crash state. Since this state is different from all other states, p "knows" that it is recovering from a crash.

2.5 Link properties

We consider links that do not create messages, or duplicate messages infinitely often. More precisely, each run $R = (F, H_{\mathcal{D}}, I, S, T)$ must satisfy the following "link properties". For all processes p and q:

- *No Creation*: If q receives a message m from p at time t, then p sent m to q before time t.
- *Finite Duplication*: If p sends a message m to q only a finite number of times, then q receives m from p only a finite number of times.

Links may intermittently drop messages, but they must satisfy the following fairness property:

- *Fair Loss*: If p sends messages to a good process q an infinite number of times, then q receives messages from p an infinite number of times.

2.6 Environments and problem solving

The correctness of an algorithm may depend on certain assumptions on the "environment", e.g., the maximum number of processes that may be bad. For example, a consensus algorithm may need the assumption that a majority of processes is good. Formally, an *environment* $\mathcal{E}$ is a set of failure patterns.

A problem P is defined by properties that sets of runs must satisfy. An algorithm A solves problem P using a failure detector $\mathcal{D}$ in environment $\mathcal{E}$ if the set of all runs $R = (F, H_{\mathcal{D}}, I, S, T)$ of A using $\mathcal{D}$ where $F \in \mathcal{E}$ satisfies the properties required by P.

Let $\mathcal{C}$ be a class of failure detectors. An algorithm A solves a problem P using $\mathcal{C}$ in environment $\mathcal{E}$ if for all $\mathcal{D} \in \mathcal{C}$, A solves P using $\mathcal{D}$ in $\mathcal{E}$. An algorithm implements $\mathcal{C}$ in environment $\mathcal{E}$ if it implements some $\mathcal{D} \in \mathcal{C}$ in $\mathcal{E}$.

3 Failure detectors for the crash-recovery model

In this section, we first consider the failure detectors that were previously proposed for solving consensus in the crash-recovery model, and then propose a new type of failure detector for this model.

3.1 Limitations of existing failure detectors

To solve consensus in the crash-recovery model, Oliveira *et al.* [10] and Hurfin *et al.* [7] assume that processes have failure detectors that output lists of processes suspected to be bad, and that these failure detectors satisfy the following property:

- *Strong Completeness*: Eventually every bad process is permanently suspected by all good processes.

Since bad processes include unstable ones, enforcing this requirement is problematic even in *synchronous* systems, as we now explain. Consider a system S in which processes take steps at perfectly synchronized rounds. In each round, a process is either up, in which case it sends a message to every process, or down, in which case it does nothing in the round. In S at most n_u processes are unstable, i.e., alternate between being up and down infinitely often. Links do not lose messages, and all messages sent in a round are received at the end of that round. In system S, it is trivial to implement a failure detector that is almost perfect: by suspecting every process from which no message was received in the current round,

[6] Note that a process cannot both access the stable storage and send a message (or issue an external output) in the same atomic step.

each process suspects exactly every process that was down in this round.

Now suppose we want to implement in S a failure detector that satisfies Strong Completeness (and possibly *only* this property). In Theorem 1, we show that any such implementation has undesirable behaviors: in some executions where *all* processes are good, some of them will eventually be suspected forever. Note that these mistakes are entirely due to the above requirement on *unstable* processes, not to the lack of synchrony.

Theorem 1. *Let $\mathcal{I}$ be any implementation of a failure detector that satisfies Strong Completeness in S. For every set of processes G of size at most n_u, there is a run of $\mathcal{I}$ in S such that (a) all processes are good, but (b) eventually all processes in G are permanently suspected by all processes in $\Pi \setminus G$.*

Intuitively, the main idea of the proof is as follows. Suppose a process u crashes for a long time. Then at some time t, to satisfy Strong Completeness, the implementation $\mathcal{I}$ is forced to suspect u (because no implementation can predict whether u will recover in the future). Suppose that u recovers after time t and stays up. If $\mathcal{I}$ keeps suspecting u forever, then this is a run in which a good process, namely u, is suspected forever (as we want to show in the theorem). Suppose, instead, that $\mathcal{I}$ trusts u again at some later point. In this case, we can crash u again for a long time. By Strong Completeness, $\mathcal{I}$ is again forced to suspect u. Now u recovers again and stays up, and either $\mathcal{I}$ keeps suspecting u forever (this is a run in which a good process u is suspected forever), or it trusts u again. We can repeat the above argument ad infinitum to obtain a run in which (a) u crashes and recovers infinitely often (it is unstable) and (b) $\mathcal{I}$ trust u an infinite number of times — a violation of Strong Completeness.

The proof of Theorem 1 follows immediately from Lemma 1 below. Note that in the round-model of execution, the only "non-determinism" is due to possible process failures and the times at which they occur. Thus, for each failure pattern F,[7] there is only *one* run of $\mathcal{I}$ in S, and we denote it by $R(F)$. A *G-crash failure pattern* is a failure pattern in which only processes in G crash.

Lemma 1. *For every set G of size at most n_u, there is a G-crash failure pattern prefix P such that the following holds. For every G-crash extension F of P in which all processes in G crash and recover at least one more time, in run $R(F)$ eventually all processes in G are permanently suspected by all processes in $\Pi \setminus G$.*

Proof. Let G be any set of size $|G| \le n_u$. Assume by contradiction that for every G-crash failure pattern prefix P, there exists a G-crash extension F of P in which all processes in G crash and recover at least one more time, such that in run $R(F)$ there is some process $p \in \Pi \setminus G$ that trusts some process $p' \in G$ infinitely often.

We now construct inductively an increasing sequence $\{P_i\}$ of failure pattern prefixes. Let P_0 be the failure pattern prefix of length 0. Given P_i, by assumption we can find a G-crash extension F_i in which all processes in G crash and recover at least one more time, such that in run $R(F_i)$ there is some process $p_i \in \Pi \setminus G$ that trusts some process $p'_i \in G$ infinitely often. Let t_i be the length of P_i and let $t_{i+1} > t_i$ be some time such that between times t_i and t_{i+1} in $R(F_i)$: (1) each process in G crashes and recovers at least once and (2) p_i trusts p'_i at least once. We define P_{i+1} to be the prefix of F_i of length t_{i+1}.

Define $P := \lim_{i\to\infty} P_i$. Then in $R(P)$, every process in G crashes an infinite number of times, no process in $\Pi \setminus G$ crashes, and some process in $\Pi \setminus G$ trusts some process in G an infinite number of times. This violates the Strong Completeness property of $\mathcal{I}$.

3.2 Failure detectors with epoch numbers

Theorem 1 shows that if we require Strong Completeness then incorrect suspicions are inevitable even in synchronous systems. Although many algorithms are designed to tolerate such failure detector mistakes, the erroneous suspicions of some good processes may hurt the performance of these algorithms. For example, the erroneous suspicions of good coordinators can delay the termination of the consensus algorithms in [3, 4, 10, 7]. Thus, requiring Strong Completeness should be avoided if possible.

In this section, we propose a new type of failure detectors that are well-suited to the crash-recovery model: Although they do not require unstable processes to be eventually suspected forever, they do provide enough information to cope with unstable processes.

At each process p, the output of such a failure detector consists of two items, $\langle trustlist, epoch\rangle$, where *trustlist* is a set of processes and *epoch* is a vector of integers indexed by the elements of *trustlist*. Intuitively, $q \in trustlist$ if p believes that q is currently up, and $epoch[q]$ is p's rough estimate of how many times q crashed and recovered so far (it is called the *epoch number of q at p*). Let $H(p,t)$ denote the output of p's failure detector module at time t. If $q \in H(p,t).trustlist$, we say that *p trusts q at time t*, otherwise we say that *p suspects q at time t*.

We define $\diamond\mathcal{S}_e$ to be the class of failure detectors $\mathcal{D}$ that satisfy the following properties:

- *Monotonicity*: At every good process, eventually the epoch numbers are nondecreasing[8]. More precisely:

  $$\forall F, \forall H \in \mathcal{D}(F), \forall g \in good(F), \forall p \in \Pi, \exists T \in \mathcal{T},$$
  $$\forall t, t' > T : [p \in H(g,t).trustlist \wedge$$
  $$p \in H(g,t').trustlist \wedge t < t'] \Rightarrow$$
  $$H(g,t).epoch[p] \le H(g,t').epoch[p]$$

- *Completeness*: For every bad process b and for every good process g, either eventually g permanently suspects b or b's epoch number at g is unbounded. More precisely:

  $$\forall F, \forall H \in \mathcal{D}(F), \forall b \in bad(F), \forall g \in good(F):$$
  $$[\exists T \in \mathcal{T}, \forall t > T, b \notin H(g,t).trustlist] \vee$$

[7] In the round-model S, a failure pattern indicates for each round which processes are up and which ones are down; a process crashes in round k, if it is up in round $k-1$ and down in round k; a process recovers in round k, if it is down in round $k-1$ and up in round k.

[8] We require the monotonicity of epoch numbers to hold only *eventually* and only at *good* processes so that the failure detector can be implemented *without* stable storage.

$$[\forall M \in \mathbf{N}, \exists t \in \mathcal{T}, b \in H(g,t).\mathit{trustlist} \wedge$$
$$H(g,t).\mathit{epoch}[b] > M]$$

- *Accuracy*: For some good process K and for every good process g, eventually g permanently trusts K and K's epoch number at g stops changing. More precisely:

$$\forall F, \forall H \in \mathcal{D}(F), \exists K \in \mathit{good}(F), \forall g \in \mathit{good}(F),$$
$$\exists M \in \mathbf{N}, \exists T \in \mathcal{T}, \forall t > T : K \in H(g,t).\mathit{trustlist} \wedge$$
$$H(g,t).\mathit{epoch}[K] = M$$

A simple implementation of $\diamond\mathcal{S}_e$ for some models of partial synchrony is given in Appendix B. This implementation does not have the limitations associated with Strong Completeness. Moreover, it does not use stable storage.

Note that $\diamond\mathcal{S}_e$ imposes requirements only on the failure detector modules of good processes. In particular, the accuracy property of $\diamond\mathcal{S}_e$ allows *unstable* processes to suspect all good processes. This is problematic because unstable processes can continue to take steps, and their incorrect suspicions may hinder the progress of some algorithms. Thus, we extend the accuracy property so that it also applies to unstable processes, as follows:

- *Strong Accuracy*: For some good process K: (a) for every good process g, eventually g permanently trusts K and K's epoch number at g stops changing; and (b) for every unstable process u, eventually whenever u is up, u trusts K and K's epoch number at u stops changing. More precisely:

$$\forall F, \forall H \in \mathcal{D}(F), \exists K \in \mathit{good}(F) : [\forall p \in \mathit{good}(F),$$
$$\exists M \in \mathbf{N}, \exists T \in \mathcal{T}, \forall t > T, \in H(p,t).\mathit{trustlist} \wedge$$
$$H(p,t).\mathit{epoch}[K] = M] \wedge [\forall u \in \mathit{unstable}(F),$$
$$\exists M \in \mathbf{N}, \exists T \in \mathcal{T}, \forall t > T, u \notin F(t) \Rightarrow$$
$$K \in H(u,t).\mathit{trustlist} \wedge H(u,t).\mathit{epoch}[K] = M]$$

The class of failure detectors that satisfy Monotonicity, Completeness, and Strong Accuracy is denoted $\diamond\mathcal{S}_u$. For convenience, we sometimes use $\diamond\mathcal{S}_e$ or $\diamond\mathcal{S}_u$ to refer to an arbitrary member of the corresponding class.

$\diamond\mathcal{S}_e$ and $\diamond\mathcal{S}_u$ are closely related: In Sect. 10 we show that one can transform $\diamond\mathcal{S}_e$ into $\diamond\mathcal{S}_u$ provided that a majority of processes are good (this transformation does not require stable storage).

4 Consensus with crash-recovery

With consensus, each process proposes a value and processes must reach a unanimous decision on one of the proposed values. The following properties must be satisfied:

- *Uniform Validity*: If a process decides v then some process previously proposed v.
- *Agreement*: Good processes do not decide different values.
- *Termination*: If all good processes propose a value, then they all eventually decide.

A stronger version of consensus, called *uniform consensus* [9], requires:

- *Uniform Agreement*: Processes do not decide different values.

The above specification allows a process to decide more than once. However, with Agreement, a good process cannot decide two different values. Similarly, with Uniform Agreement, no process (whether good or bad) can decide two different values.

The algorithms that we provide solve uniform consensus, and the lower bounds that we prove hold even for consensus.

When processes have access to stable storage, a process proposes v, or decides v, by writing v into corresponding local stable storage locations. By checking these locations, a process that recovers from a crash can determine whether it previously proposed (or decided) a value.

When processes do not have access to stable storage, proposing and deciding v occur via an external input and output containing v, and so when a process recovers it cannot determine whether it has previously proposed or decided a value. Thus it is clear that if stable storage is not available and *all* processes may crash and recover, consensus cannot be solved. In many systems, however, it is reasonable to assume that in each execution of consensus there is a minimum number of processes that do not crash. In such systems, consensus *is* solvable without stable storage provided certain conditions are met, as we will see next.

5 On the necessity of stable storage for consensus

In this section, we determine some necessary conditions for solving consensus without stable storage. Consider a system in which at least n_a processes are always-up and at most n_b are bad. Our first result is that if $n_a \leq n_b$ then it is impossible to solve consensus without stable storage, even in systems where there are no unstable processes, links are reliable, and processes can use an *eventually perfect failure detector* $\diamond\mathcal{P}$. Informally, for the crash-recovery model, $\diamond\mathcal{P}$ outputs a tag $\in \{\mathrm{AU}, \mathrm{EU}, \mathrm{UN}, \mathrm{ED}\}$ for each process such that:

- There is a time after which at each process the tag of every process p is AU, EU, UN, or ED iff p is always-up, eventually-up, unstable, or eventually-down, respectively.

Note that $\diamond\mathcal{P}$ is stronger than the other failure detectors in this paper and in [10, 7].

Theorem 2. *If $n_a \leq n_b$ consensus cannot be solved without stable storage even in systems where there are no unstable processes, links do not lose messages, and processes can use $\diamond\mathcal{P}$.*

This result is tight in the sense that if $n_a > n_b$ then we *can* solve consensus without stable storage using a failure detector that is weaker than $\diamond\mathcal{P}$ (see Sect. 6).

The impossibility result of Theorem 2 assumes that processes do not use any stable storage at all. Thus, if a process crashes it cannot "remember" its previous proposal and/or decision value. Suppose stable storage is available, but to minimize the cost of accessing it, we want to use it *only* for storing (and retrieving) the proposed and decision values. Is $n_a > n_b$ still necessary to solve consensus? It turns out that if $n_b > 2$, the answer is yes:

Theorem 3. *Suppose that each process can use stable storage only for storing and retrieving its proposed and decision*

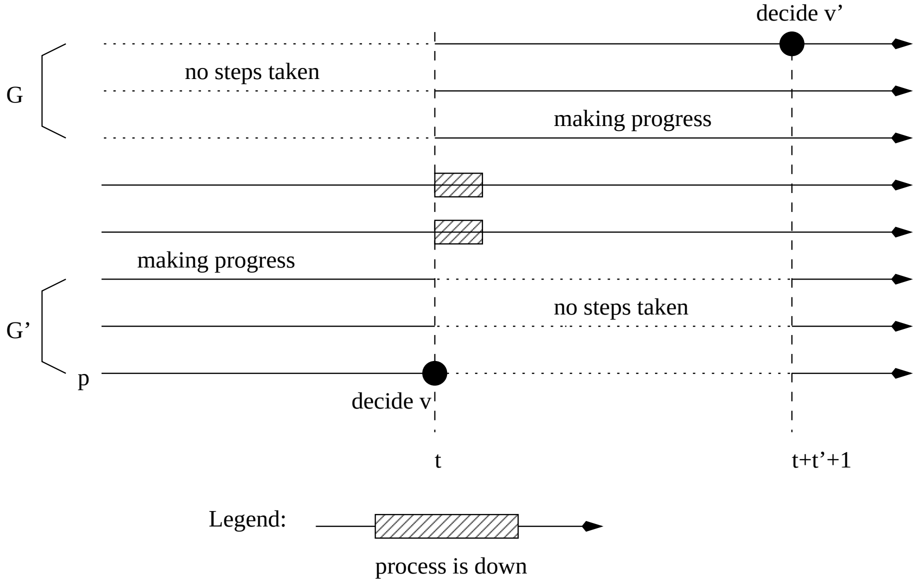


Fig. 1. A run constructed to prove Theorem 3

values. If $n_a \leq n_b$ and $n_b > 2$ then consensus cannot be solved even in systems where there are no unstable processes, links do not lose messages, and processes can use $\diamond\mathcal{P}$.

Theorems 2 and 3 have similar proofs, and so we only give a detailed proof of Theorem 3 here. The main idea of this proof is as follows. For a contradiction, assume that there is a consensus algorithm that does not use stable storage (except for saving the proposed and decision values) and works for $n_a \leq n_b$ and $n_b > 2$. Let G be a subset of n_b processes. Consider a run in which initially processes in G are very slow, i.e., they do not take steps for a while (see Fig. 1). From the point of view of the other processes, all the processes in G could be bad, so eventually some process p not in G decides some value v. Let t be the time when p decides v, and let G' be a subset of n_b processes that contains p and is disjoint from G. At time t, every process that is not in G or G' crashes and recovers (i.e., it loses its intermediate state and restarts in a recovery state; at this point it "remembers" only its own proposed value). Note that at time t, processes not in G' do not know about the decision value v (and processes in G have not taken any step yet). From time t, all messages still in transit at time t are delayed, and all the processes in G' stop taking steps for a long time. From the point of view of processes not in G', it could be that: (1) all the n_b processes in G' are bad, (2) all the processes not in G or G' are eventually up, and (3) all the n_b processes in G are always up (this scenario is consistent with the assumption that $n_a \leq n_b$). Thus, the processes not in G' must decide without input from G', and in particular without the knowledge that p has decided v. Let v' be the decision of the processes not in G'.

It remains to show that v could be different from v' (a contradiction). Proving this is not simple because: (1) the processes not in G or G' participate in the decision of both v and v', (2) for both decisions, they propose the same values (each process stores its proposed value in stable storage, and so when it recovers it proposes the same value), and (3) the processes not in G or G' could be a (large) majority of the processes. By Lemma 2, however, we can indeed find some assignment of proposed values to processes, and sets G and G', such that $v \neq v'$ (this lemma uses the fact that $n_b > 2$).

We now prove Theorem 3 in detail.

Consider a system with $n_a \leq n_b$, $n_b > 2$, and such that links do not lose messages. Assume for a contradiction that there is a consensus algorithm $\mathcal{A}$ that (1) uses stable storage only for storing and retrieving its proposed and decision values; and (2) uses failure detector $\diamond\mathcal{P}$. Henceforth, in all runs of $\mathcal{A}$ that we consider, processes always propose a value in $\{0, 1\}$ at the beginning of the run.

Definition 1. *Let R be a set of runs of $\mathcal{A}$. By the properties of consensus, in every run in R, all good processes eventually decide the same value. We say that R is* 0-valent *(resp.* 1-valent*) if in every run in R, the good processes decide* 0 *(resp.* 1*). We say that R is* bivalent *if R is neither 0-valent nor 1-valent, i.e., R has a run in which the good processes decide* 0 *and a run in which the good processes decide* 1.

In the next definitions, let V be an assignment of proposed values, one for each process, and G_{au} and G_{bad} be disjoint subsets of size n_b of processes.

Definition 2. *$R(V, G_{bad})$ is defined to be the set of runs of $\mathcal{A}$ such that (1) the value proposed by each process is given by V; (2) processes in G_{bad} crash at the beginning and never recover; and (3) processes not in G_{bad} never crash.*

Note that in any run in $R(V, G_{bad})$, processes in G_{bad} are bad, and the other processes are always-up.

Definition 3. *$R(V, G_{au}, G_{bad})$ is defined to be the set of runs of $\mathcal{A}$ such that (1) the value proposed by each process is given by V; (2) processes in G_{bad} crash at the beginning and never recover; (3) processes in G_{au} never crash; and (4) processes not in $G_{au} \cup G_{bad}$ crash at the beginning, recover right afterwards, and never crash again.*

Note that in any run in $R(V, G_{au}, G_{bad})$, processes in G_{au} are always-up,[9] processes in G_{bad} are bad, and the other processes are eventually-up.

Lemma 2. *There exists V and disjoint subsets of processes G and G' of size n_b such that (1) in some run $r \in R(V, G)$, the first good process to decide is in G'; (2) in some run $r' \in R(V, G, G')$, the decision value of the good processes is different from the decision value of the good processes in r.*

Proof. There are two cases. For the first case, assume that there is V and a set G of size n_b such that $R(V, G)$ is bivalent. Then, for $i = 0, 1$ we can find a run r_i in $R(V, G)$ in which good processes decide i. Let p_i be the first good process to decide in r_i and let G' be any subset of size n_b that is disjoint from G and contains p_0 and p_1. Let r' be any run in $R(V, G, G')$. If good processes in r' decide 0, let $r := r_1$; else let $r := r_0$. Then clearly r and r' satisfy conditions (1) and (2) of the lemma.

For the other case, assume that for every $\bar{V}$ and every set $\bar{G}$ of size n_b, $R(\bar{V}, \bar{G})$ is either 0-valent or 1-valent. Let $G = \{n - n_b + 1, \ldots, n\}$. For $i = 0, 1, \ldots, n$, let V_i be the assignment of proposed values such that the proposed value for processes $1, 2, \ldots, i$ is 1, and for processes $i + 1, \ldots, n$, it is 0. Then clearly $R(V_0, G)$ is 0-valent. Moreover, for any run in $R(V_{n-n_b}, G)$, all processes that ever take any steps propose 1, so $R(V_{n-n_b}, G)$ is 1-valent. Therefore, for some $j \in \{0, \ldots, n - n_b - 1\}$ we have that $R(V_j, G)$ is 0-valent and $R(V_{j+1}, G)$ is 1-valent.

Let $r_0 \in R(V_j, G)$ and $r_1 \in R(V_{j+1}, G)$. Note that good processes in r_0 decide 0, and in r_1 good processes decide 1. For $i = 0, 1$, let p_i be the first good process to decide in r_i and let G' be any subset of size n_b that is disjoint from G and contains p_0, p_1 and $j + 1$ (here we are using the fact that $n_b > 2$). Note that the only difference between V_j and V_{j+1} is the proposed value of process $j + 1$. Moreover, $j + 1 \in G'$, so that process $j + 1$ never takes any steps in any runs in $R(V_j, G, G')$ or in $R(V_{j+1}, G, G')$. Therefore, $R(V_j, G, G') = R(V_{j+1}, G, G')$. Let $r' \in R(V_j, G, G')$. If good processes in r' decide 0, let $r := r_1$ and $V := V_{j+1}$; otherwise, let $r := r_0$ and $V := V_j$. Then clearly r and r' satisfy conditions (1) and (2) of the lemma.

Proof of Theorem 3 (Sketch). Let V, G, G', r and r' be as in Lemma 2. Let p (resp. p') be the first good process to decide in r (resp. r'), let t (resp. t') be the time when this decision happens and let v (resp. v') be the decision value. Then $v \neq v'$ and $p \in G'$. We now construct a new run r'' of $\mathcal{A}$ as follows. The proposed value of processes is given by V. Initially processes in G do not take any steps and processes in $\Pi \setminus G$ behave as in run r. This goes on until time t (when p decides v). Messages sent and not received by time t are delayed until after time $t + t' + 1$. At time $t + 1$, all processes in G' stop taking steps, and processes not in $G \cup G'$ crash. At time $t + 2$, processes not in $G \cup G'$ recover. Note that at time $t + 2$, the state of all processes not in G' are as in run r' at time 1 (this is because processes could not use stable storage to keep intermediate states of the computation). From time $t + 2$ to time $t + t' + 1$, processes not in G' behave as in run r' from time 1 to t'. Thus, at time $t + t' + 1$, process p' decides v'. After time $t + t' + 1$, (1) all processes take steps in a round-robin fashion, (2) all messages ever sent are received, (3) the failure detector behaves perfectly, i.e., at every process the tag of processes in $G \cup G'$ is AU and the tag of processes not in $G \cup G'$ is EU.

Note that r'' is a run of $\mathcal{A}$ in which all processes are good. Moreover, p decides v and p' decides $v' \neq v$. This violates the agreement property of consensus. □

We now briefly outline the proof of Theorem 2. Let $n_a \leq n_b$. If $n_b = 0$ then $n_a = 0$ and it is easy to see that there can be no consensus algorithm (since all processes may lose their proposed values by crashing at the beginning). So let $n_b > 0$. Assume for a contradiction that there is a consensus algorithm $\mathcal{A}$ that does not use stable storage (and uses $\diamond\mathcal{P}$).

Lemma 3. *There exists V and disjoint subsets of processes G and G' of size n_b such that the decision value of good processes in some run $r \in R(V, G)$ is different from the decision value of good processes in some run $r' \in R(V, G, G')$.*

Proof. The proof is similar to the proof of Lemma 2.

The rest of the proof of Theorem 2 uses Lemma 3 and otherwise is similar to the proof of Theorem 3.

6 Solving consensus without stable storage

It turns out that if $n_a > n_b$, consensus can be solved without stable storage using $\diamond\mathcal{S}_e$. This is somewhat surprising since $n_a > n_b$ allows a majority of processes to crash (and thus lose all their states). Note that the requirement of $n_a > n_b$ is "tight": in the previous section, we proved that if $n_a \leq n_b$ consensus cannot be solved without stable storage even with $\diamond\mathcal{P}$, a failure detector that is stronger than $\diamond\mathcal{S}_e$.

The consensus algorithm that uses $\diamond\mathcal{S}_e$ is given in Appendix A. In this section, we present a more efficient algorithm that uses a minor variant of $\diamond\mathcal{S}_e$, denoted $\diamond\mathcal{S}'_e$. The only difference between $\diamond\mathcal{S}_e$ and $\diamond\mathcal{S}'_e$ is that while the accuracy property of $\diamond\mathcal{S}_e$ requires that K be a *good* process (see Sect. 3.2), the accuracy property of $\diamond\mathcal{S}'_e$ additionally requires that K be an *always-up* process if such a process exists. It is worth noting that the implementation of $\diamond\mathcal{S}_e$ in Appendix B also implements $\diamond\mathcal{S}'_e$.

The consensus algorithm that we give here always satisfies the Uniform Agreement and Validity properties of uniform consensus for any choice of n_a and n_b, and if $n_a > n_b$ then it also satisfies the Termination property.

This algorithm, shown in Fig. 2, is based on the rotating coordinator paradigm [3] and uses $\diamond\mathcal{S}'_e$. It must deal with unstable processes and link failures. More importantly, since more than half of the processes may crash and completely lose their states, and then recover, it must use new mechanisms to ensure the "locking" of the decision value (so that successive coordinators do not decide differently).[10] We first explain how the algorithm deals with unstable processes and link failures, and then describe the algorithm and the new mechanisms for locking the decision value.

How does a rotating coordinator algorithm cope with an unstable coordinator? In [10,7] the burden is entirely on the

[9] This is possible because $|G_{au}| = n_b \geq n_a$.

[10] The standard technique for locking a value is to ensure that a majority of processes "adopt" that value. This will not work here: a majority of processes may crash and recover, and so *all* the processes that adopted a value may later forget the value they adopted.

For every process p:

```
Initialization:
  $R_p \leftarrow \emptyset$; $decisionvalue_p \leftarrow \bot$; for all $q \in \Pi \setminus \{p\}$ do $xmitmsg[q] \leftarrow \bot$
To s-send $m$ to $q$:
  if $q \neq p$ then $xmitmsg[q] \leftarrow m$; send $m$ to $q$ else simulate receive $m$ from $p$
Task retransmit:
  repeat forever
    for all $q \in \Pi \setminus \{p\}$ do if $xmitmsg[q] \neq \bot$ then send $xmitmsg[q]$ to $q$
upon receive $m$ from $q$ do
  if $m = \text{RECOVERED}$ then $R_p \leftarrow R_p \cup \{q\}$
  if $m = (decisionvalue, \text{DECIDE})$ and $decisionvalue_p = \bot$ then
    $decisionvalue_p \leftarrow decisionvalue$; decide$(decisionvalue_p)$
    terminate task {skip_round, 4phases, participant, coordinator, retransmit}
  if $m \neq (-, \text{DECIDE})$ and $decisionvalue_p \neq \bot$ then send $(decisionvalue_p, \text{DECIDE})$ to $q$
upon propose$(v_p)$:                              {$p$ proposes $v_p$ via an external input containing $v_p$}
  $(r_p, estimate_p, ts_p) \leftarrow (1, v_p, 0)$; fork task {4phases, retransmit}
Task 4phases:
  $c_p \leftarrow (r_p \bmod n) + 1$; fork task {skip_round, participant}
  if $p = c_p$ then fork task coordinator

Task coordinator:
  {Stage 1: Phase NEWROUND}
    $c\_seq_p \leftarrow 0$
    repeat
      $PrevR_p \leftarrow R_p$; $c\_seq_p \leftarrow c\_seq_p + 1$
      s-send $(r_p, c\_seq_p, \text{NEWROUND})$ to all
      wait until [ received $(r_p, c\_seq_p, estimate_q,$
          $ts_q, \text{ESTIMATE})$ from
          $\max(n_b + 1, n - n_b - |R_p|)$ processes ]
    until $R_p = PrevR_p$
    $t \leftarrow$ largest $ts_q$ such that $p$ received
      $(r_p, c\_seq_p, estimate_q, ts_q, \text{ESTIMATE})$
    $estimate_p \leftarrow$ select one $estimate_q$ such that
      $p$ received $(r_p, c\_seq_p, estimate_q, t, \text{ESTIMATE})$
    $ts_p \leftarrow r_p$
  {Stage 2: Phase NEWESTIMATE}
    $c\_seq_p \leftarrow 0$
    repeat
      $PrevR_p \leftarrow R_p$; $c\_seq_p \leftarrow c\_seq_p + 1$
      s-send $(r_p, c\_seq_p, estimate_p,$
          $\text{NEWESTIMATE})$ to all
      wait until [ received $(r_p, c\_seq_p, \text{ACK})$ from
          $\max(n_b + 1, n - n_b - |R_p|)$ processes ]
    until $R_p = PrevR_p$
    s-send $(estimate_p, \text{DECIDE})$ to all

Task participant:
  {Stage 1: Phase ESTIMATE}
    s-send $(r_p, \text{WAKEUP})$ to $c_p$
    $max\_seq_p \leftarrow 0$
    repeat
      if received $(r_p, seq, \text{NEWROUND})$ from $c_p$
        for some $seq > max\_seq_p$ then
        s-send $(r_p, seq, estimate_p, ts_p, \text{ESTIMATE})$ to $c_p$
        $max\_seq_p \leftarrow seq$
    until [ received $(r_p, seq, estimate_{c_p}, \text{NEWESTIMATE})$
      from $c_p$ for some $seq$ ]
    if $p \neq c_p$ then
      $(estimate_p, ts_p) \leftarrow (estimate_{c_p}, r_p)$

  {Stage 2: Phase ACK}
    $max\_seq_p \leftarrow 0$
    repeat forever
      if received $(r_p, seq, estimate_{c_p}, \text{NEWESTIMATE})$
        from $c_p$ for some $seq > max\_seq_p$ then
        s-send $(r_p, seq, \text{ACK})$ to $c_p$
        $max\_seq_p \leftarrow seq$

Task skip_round:
  $d \leftarrow \mathcal{D}_p$                                           {query $\Diamond\mathcal{S}'_e$}
  if $c_p \in d.trustlist \setminus R_p$ then
    repeat $d' \leftarrow \mathcal{D}_p$                                 {query $\Diamond\mathcal{S}'_e$}
    until [ $c_p \notin d'.trustlist \setminus R_p$ or $d.epoch[c_p] < d'.epoch[c_p]$ or received some message $(r, \ldots)$ such that $r > r_p$ ]
  terminate task {4phases, participant, coordinator}                     {abort current round}
  repeat $d \leftarrow \mathcal{D}_p$ until $d.trustlist \setminus R_p \neq \emptyset$          {query $\Diamond\mathcal{S}'_e$}
  $r_p \leftarrow$ the smallest $r > r_p$ such that $[(r \bmod n) + 1] \in d.trustlist \setminus R_p$ and $r \geq \max\{r' \mid p \text{ received } (r', \ldots)\}$
  fork task 4phases                                                      {go to a higher round}
upon recovery:
  $decisionvalue_p \leftarrow \bot$; for all $q \in \Pi \setminus \{p\}$ do $xmitmsg[q] \leftarrow \bot$; fork task retransmit
  s-send RECOVERED to all
```

Fig. 2. Solving Consensus without Stable Storage using $\Diamond\mathcal{S}'_e$

failure detector: it is postulated that every unstable process is eventually suspected forever. In our algorithm, the failure detector is not required to suspect unstable processes: they can be trusted as long as their epoch number increases from time to time — a requirement that is easy to enforce. If the epoch number of the current coordinator increases at a process, this process simply abandons this coordinator and goes to another one.

To deal with the message loss problem, each process p has a task *retransmit* that periodically retransmits the last message sent to each process (only the last message really matters, just as in [4,6,7]). This task is terminated once p decides.

We now describe the algorithm in more detail. When a process recovers from a crash, it stops participating in the algorithm, except that it periodically broadcasts a RECOVERED message until it receives the decision value. When a process p receives a RECOVERED message from q, it adds q to a set R_p of processes known to have recovered.

Processes proceed in asynchronous rounds, each one consisting of two stages. In the first stage, processes send a WAKEUP message to the coordinator c so that c can start the current round (if it has not done so yet). The coordinator c broadcasts a NEWROUND message to announce a new round, and each process sends its current estimate of the decision value — together with a timestamp indicating in which round it was obtained — to c. Then c waits for estimates from $\max(n_b+1, n-n_b-|R_c|)$ processes — this is the maximum number of estimates that c can wait for without fear of blocking forever, because more than n_b processes are always-up and respond, and at most $n_b + |R_c|$ processes have crashed and do not respond. Then c checks whether during the collection of estimates it detected the recovery of a process that never recovered before ($R_c \neq \mathit{PrevR}_c$). If so, c restarts the first stage from scratch.[11] Otherwise, c chooses the estimate with the largest timestamp as its new estimate and proceeds to the second stage.

In the second stage, c broadcasts its new estimate; when a process receives this estimate, it changes its own estimate and sends an ACK to c. Process c waits for ACK messages from $\max(n_b+1, n-n_b-|R_c|)$ processes. As before, c restarts this stage from scratch if during the collection of ACKs it detected the recovery of a process that never recovered before ($R_c \neq \mathit{PrevR}_c$). Finally c broadcasts its estimate as the decision value and decides accordingly. Once a process decides, it enters a passive state in which, upon receipt of a message, the process responds with the decision value.

A round r can be interrupted by task *skip_round* (which runs in parallel with tasks *coordinator* and *participant*): a process p aborts its execution of round r if (1) it suspects the coordinator c of round r, or (2) it trusts c but detects an increase in the epoch number of c, or (3) it detects a recovery of c, or (4) it receives a message from a round $r' > r$. When p aborts round r, it jumps to the lowest round $r' > r$ such that (1) p trusts the coordinator c' of round r', (2) p has not detected a recovery of c' ($c' \notin R_p$) and (3) p has not (yet) received any message with a round number higher than r'.

The code in lines 31–33 is executed atomically, i.e., it cannot be interrupted, except by a crash. As an obvious optimization, the coordinator of round 1 can skip phase NEWROUND and simply set its estimate to its own proposed value. We omit this optimization from the code.

The correctness of the algorithm relies on the following crucial property: if the coordinator sends a decision for v in some round, then value v has previously been "locked", i.e., in any later round, a coordinator can only choose v as its new estimate. This property is ensured by two mechanisms: (1) the coordinator uses $\max(n_b+1, n-n_b-|R_p|)$ as a threshold to collect estimates and ACKs, and (2) the coordinator restarts the collection of estimates and ACKs from scratch if it detects a new recovery ($R_c \neq \mathit{PrevR}_c$).

The importance of mechanism (2) is illustrated in Fig. 3: it shows a bad scenario (a violation of the crucial property above) that could occur if this mechanism is omitted. The system consists of four processes $\{c, p, p', c'\}$. Assume that $n_b = 1$ and there are at least $n_a = 2$ processes that are always up. At point A, the coordinator c of round r sends its estimate 0 to all, and at B, it receives ACKs from itself and p. At F, p' recovers from a crash and sends a RECOVERED message to all. At G, c has received one RECOVERED message from p' (so $|R_c| = 1$) and two ACKs. Since $\max(n_b+1, n-n_b-|R_c|) = 2$, c completes its collection of ACKs (this is the maximum number of ACKs that c can wait for without fear of blocking), and c sends a decision for 0 to all in round r. Meanwhile, at C, p recovers from a crash and sends a RECOVERED message to all, and c' receives this message before D. At D, c' becomes the coordinator of round $r' > r$ and sends a NEWROUND message to all. At E, c' has received two estimates for 1, one from itself and one from p'. Since it has also received one RECOVERED message from p, c' completes its collection of estimates, and chooses 1 as its new estimate for round r' — even though c sends a decision for 0 in an earlier round.

The proof of the algorithm shows that mechanism (2) prevents this and other similar bad scenarios. In this example, if c had used mechanism (2), then at G it would have restarted the collection of ACKs from scratch because $\mathit{PrevR}_c = \emptyset \neq \{p'\} = R_c$.[12]

Theorem 4. *The algorithm of Fig. 2 satisfies the Uniform Validity and Uniform Agreement properties of uniform consensus. If at most n_b processes are bad, and more than n_b processes are always up, then it also satisfies the Termination property.*

The proof follows.

Definition 4. *We say that p* is in round r at time t *if p does not crash by time t and the value of variable r_p at time t is r. A process p* starts round r *when p assigns r to variable r_p.*

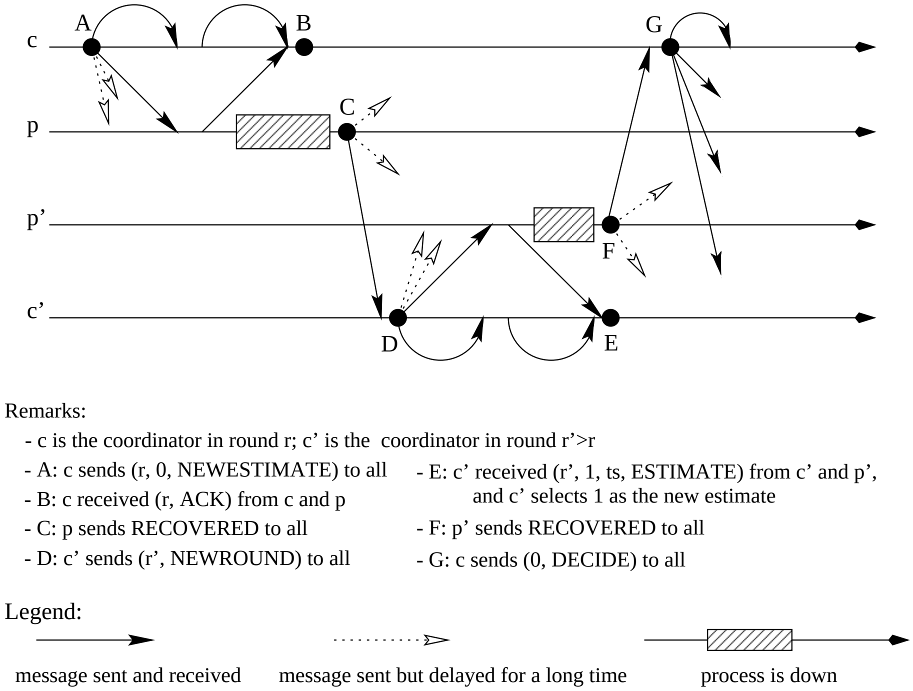
Lemma 4 (Uniform Validity). *If a process decides v then some process previously proposed v.*

Proof. Trivial. □

Lemma 5. *Suppose that in some round r the coordinator c* s-sends $(est,$ DECIDE$)$ *in line 43. In every round $r' \geq r$, if the coordinator c' selects a new estimate value est' in line 31, then $est = est'$.*

[11] An obvious optimization is for c to check *during the collection of estimates* whether $R_c \neq \mathit{PrevR}_c$. If so it can restart the first stage right away.

[12] It is not sufficient to use the restarting mechanism only for collecting ACKs: a symmetric example shows that this mechanism must also be used for collecting estimates.



Remarks:

- c is the coordinator in round r; c' is the coordinator in round r'>r
- A: c sends (r, 0, NEWESTIMATE) to all
- B: c received (r, ACK) from c and p
- C: p sends RECOVERED to all
- D: c' sends (r', NEWROUND) to all
- E: c' received (r', 1, ts, ESTIMATE) from c' and p', and c' selects 1 as the new estimate
- F: p' sends RECOVERED to all
- G: c sends (0, DECIDE) to all

Legend:

message sent and received

message sent but delayed for a long time

process is down

Fig. 3. A bad scenario that can occur if mechanism (2) is not used

Proof. The proof is by induction on the round number r'. The claim trivially holds for $r' = r$. Now assume that the claim holds for all $r', r \leq r' < k$. Let c' be the coordinator of round k. We will show that the claim holds for $r' = k$, i.e., if c' selects a new estimate value est' in line 31 in round k, then $est' = est$.

Since c s-sends $(est,$ DECIDE$)$ in line 43 in round r, c executes the wait statement in line 40 only finitely often in round r. Similarly, since c' executes line 31 in round k, c' executes the wait statement in line 25 only finitely often in round k. Thus the following definitions are valid:

- seq_A, the value of c_seq_c just after c executes the wait statement in line 40 for the last time in round r.
- A, the subset of processes from which c has received (r, seq_A, ACK) by the time c exits the wait statement in line 40 for the last time in round r.
- R_A, the value of set R_c just after c executes the wait statement in line 40 for the last time in round r.
- seq_E, the value of $c\_seq_{c'}$ just after c' executes the wait statement in line 25 for the last time in round k.
- E, the subset of processes from which c' has received messages of the form (k, seq_E, *, *, ESTIMATE) by the time c' exits the wait statement in line 25 for the last time in round k.
- R_E, the value of set $R_{c'}$ just after c' executes the wait statement in line 25 for the last time in round k.

We first claim that (1) processes in R_A crash before c starts s-sending (r, seq_A, est, NEWESTIMATE) to any process in round r (line 38). Indeed, just after c executes line 40 for the last time in round r, we have that $R_A = R_c$ (by the definition of R_A) and $R_c = PrevR_c$ (by the condition in line 42). Therefore, $PrevR_c = R_A$. But all processes in $PrevR_c$ crashed before c starts s-sending (r, seq_A, est, NEWESTIMATE). So the claim follows.

Note that (2) $|A| \geq n_b + 1$ (this is due to the guard in line 40). We now show that $A \cap R_A = \emptyset$. By the previous claim, if a process $p \in R_A$ then p crashes before c starts s-sending (r, seq_A, est, NEWESTIMATE), which happens before any process s-sends (r, seq_A, ACK) to c. So p crashes before any process s-sends (r, seq_A, ACK) to c. Since after a process crashes (and recovers) it can s-send only RECOVERED or DECIDE messages, it follows that $p \notin A$. Thus, $A \cap R_A = \emptyset$.

So, $|A \cup R_A| = |A| + |R_A|$. By the threshold used to collect ACKs in lines 40-41, we have $|A| \geq \max(n_b + 1, n - n_b - |R_A|)$, and thus (3) $|A \cup R_A| \geq n - n_b$.

By analogous arguments we can show that (4) $|E| \geq n_b + 1$ and (5) $|E \cup R_E| \geq n - n_b$.

We now show that $E \cap A \neq \emptyset$. Suppose, for contradiction, that $E \cap A = \emptyset$. By (3) and (4), we have $E \cap (A \cup R_A) \neq \emptyset$. Since $E \cap A = \emptyset$, we have $E \cap R_A \neq \emptyset$. Let $p \in E \cap R_A$. Clearly, c' starts s-sending (k, seq_E, NEWROUND) to processes before p receives such a message, which happens before p s-sends a message of the form (k, seq_E, *, *, ESTIMATE) to c' (p s-sends such message because $p \in E$), which happens before p crashes (since after a process crashes and recovers, it can s-send only RECOVERED or DECIDE messages), which happens before c starts s-sending (r, seq_A, est, NEWESTIMATE) to processes (this follows from the fact that $p \in R_A$ and Claim (1)). From all this, we conclude that c' starts s-sending (k, seq_E, NEWROUND) before c starts s-sending (r, seq_A, est, NEWESTIMATE).

By (2) and (5), we have $A \cap (E \cup R_E) \neq \emptyset$. By an argument analogous to the above one, we can conclude that c starts s-sending (r, seq_A, est, NEWESTIMATE) before c' starts s-sending (k, seq_E, NEWROUND). This is a contradiction. Hence, we conclude that $E \cap A \neq \emptyset$.

Let $p \in E \cap A$. By the definition of A, p s-sends (r, seq_A, ACK) to c in round r. Before doing so, p updates ts_p to

r (line 33 or 56). By the definition of E, for some est'' and ts'', p s-sends (k, seq_E, est'', ts'', ESTIMATE) to c' in line 51 in round k. Since $k > r$ and the value of ts_p is nondecreasing, we have $ts'' \geq r$. Moreover, it is easy to see that c' does not receive any messages of the form (k, *, *, ts, ESTIMATE) with $ts \geq k$. So, the timestamp t that c' selects in line 29 in round k is such that $r \leq t < k$. Let q be the process whose estimate value est' is selected in line 31 in round k. Then in round k, q s-sends (k, seq_E, est', t, ESTIMATE). We claim that in round t, q updates $estimate_q$ to est' in line 31 or 56. Indeed, in round t, q updates $estimate_q$ to some value est''' and q updates ts_q to t. After that, q does not change $estimate_q$ and ts_q before round k (because otherwise in round k, ts_q would be different from t and q would not s-sends (k, seq_E, est', t, ESTIMATE)). Therefore $est' = est'''$.

Since q updates $estimate_q$ to est' in round t (line 31 or 56), it is easy to see that the coordinator of round t selects est' as the new estimate value in line 31. By the induction hypothesis, we have $est' = est$. This shows the induction step. □

Lemma 6. *If processes c and c' s-send (est, DECIDE) and (est', DECIDE) in line 43 in rounds r and r', respectively, then $est = est'$.*

Proof. Assume without loss of generality that $r' \geq r$. Since line 43 is executed only by the coordinator, c and c' are the coordinators of rounds r and r', respectively. Since c' s-sends (est', DECIDE) in line 43 in round r', c' selects est' in line 31. By Lemma 5, $est = est'$. □

Lemma 7 (Uniform Agreement). *No two processes decide differently.*

Proof. Suppose that processes p and p' decide on values est and est', respectively. Process p decides est in line 11 after receiving message (est, DECIDE). By a simple induction, some process must have s-sent message (est, DECIDE) in line 43. Similarly, process p' decides est' in line 11, and so some process must have s-sent message (est', DECIDE) in line 43. By Lemma 6, $est = est'$. □

Lemma 8. *A process can start only finitely many rounds.*

Proof. In order to obtain a contradiction, suppose that there are processes that start infinitely many rounds. Let P be the set of all such processes. P contains only always-up processes, since a process that crashes does not start any rounds ever again (even if it recovers). For any process $p \in P$ and any round $r \geq 1$, p eventually starts a round higher than r. Let r_p^+ be the lowest round higher than r that p starts and let r_p^- be the highest round lower than or equal to r that p starts. Then $1 \leq r_p^- \leq r < r_p^+$.

By the Accuracy property of $\diamond\mathcal{S}'_e$, we can find a time T and an always-up process K such that after T, K is never suspected by any good process and the epoch number of K at every good process stops changing.

Let r be a round such that (1) K is the coordinator of round r, and (2) no process in $\Pi \setminus P$ starts a round higher than r, and (3) for every $p \in P$, p starts round r_p^- after time T. Such round clearly exists because processes in $\Pi \setminus P$ start only finitely many rounds and processes in P start infinitely many rounds.

Let p be the first process to start a round higher than r. By (2), $p \in P$ and by the definition of r_p^- and r_p^+, p selects round r_p^+ when it executes line 71 in round r_p^-. This implies that $r_p^- = r$: indeed, if $r_p^- < r$ then p does not select round r_p^+ in line 71; instead, it selects a round number that is at most r since (a) p trusts the coordinator K of round r (by (3) and the definitions of T and K), and (b) $K \notin R_p$ (since K is always-up), and (c) p does not receive any messages of a round higher than r (since p is the first process to start a round higher than r). So $r_p^- = r$. By (3), p starts round r after time T. By (1) and the definition of T and K, while p is in round r, condition $K \in d.\mathit{trustlist} \setminus R_p$ in line 66 evaluates to true and condition ($K \notin d'.\mathit{trustlist} \setminus R_p$ **or** $d.\mathit{epoch}[K] < d'.\mathit{epoch}[K]$) in line 68 always evaluates to false. Since p starts a round higher than r, it does not loop forever in lines 67–68. So p eventually receives a message of a round higher than r while in round r. This contradicts the fact that p is the first process to start a round higher than r. □

Definition 5. *We say that* an always-up process p blocks in round r *if p starts round r but p does not start a higher round, and p never decides.*

Lemma 9. *If an always-up process p blocks in round r, then in this round its skip_round task loops forever in lines 67–68.*

Proof. Clearly, while p is in round r, its task *skip_round* must loop forever in lines 67–68 or in line 70 (otherwise p starts a round higher than r). By the Accuracy property of $\diamond\mathcal{S}'_e$, p eventually trusts some always-up process c forever. Moreover $c \notin R_p$ since c never crashes. So p cannot loop forever in line 70. Therefore p loops forever in lines 67–68. □

Lemma 10. *Suppose an always-up process p proposes but never decides. If p receives a message of round r, then eventually p starts some round $r' \geq r$.*

Proof. In order to obtain a contradiction, suppose that p never starts any round $r' \geq r$. Since p proposes, p starts some round (namely, round 1). Since p does not decide, p blocks in some round $r'' < r$. By Lemma 9, while in round r'' the *skip_round* task of p loops forever in lines 67–68. Since p receives a message of round r, p eventually exits the loop in lines 67–68 — a contradiction. □

Definition 6. *We say that an eventually-up process* stabilizes at time t *if it recovers at time t and does not crash afterwards. By convention, we say that an always-up process stabilizes at time 0.*

Lemma 11. *Suppose p and q are good processes. If (1) p s-sends m to q after p stabilizes, (2) m is the last message p s-sends to q, and (3) p does not decide after p stabilizes, then q receives m from p infinitely often.*

Proof. By (1), (2) and (3), p sends m to q infinitely often in task *retransmit* (line 7). By the Fair Loss property of links, q receives messages from p infinitely often. Note that m is the only message that p sends to q infinitely often: this is because (1) in task *retransmit*, p eventually sends no message different from m to q, and (2) outside task *retransmit*, p can only send messages of the form ($*$, DECIDE) (line 13); however, such messages are sent only finitely often since p does not decide after p stabilizes. Therefore, by the No Creation and Finite Duplication properties of links, q receives from p only finitely many messages different from m. Since q receives messages from p infinitely often, it follows that q receives m from p infinitely often. □

Lemma 12. *Suppose p and q are good processes. If p decides after p stabilizes and p receives non-*DECIDE *messages from q an infinite number of times, then eventually q decides after q stabilizes.*

Proof. After p stabilizes and decides, every time p receives a non-DECIDE message from q, p sends a DECIDE message to q (line 13). Therefore p sends DECIDE messages to q infinitely often. Moreover, this is the only message that p sends to q infinitely often (since after p decides, it terminates all tasks). By the link properties, this implies that q receives DECIDE messages from p infinitely often. Thus, eventually q decides after q stabilizes. □

Lemma 13. *If an always-up process p blocks in a round r, then the coordinator c of this round is also an always-up process. Moreover, if $p \neq c$ then c receives messages of round r from p infinitely often.*

Proof. Note that if $p = c$ then the lemma holds trivially. So assume that $p \neq c$. We first prove that c is a good process. In order to obtain a contradiction, suppose that c is bad. By the Completeness and Monotonicity properties of $\diamond\mathcal{S}'_e$, eventually either c is suspected by p forever, or the epoch number of c at p is nondecreasing and unbounded. Therefore, in round r, p eventually exits the loop in lines 67–68. This contradicts Lemma 9. So c is a good process.

We now claim that c receives messages of round r from p infinitely often. To show the claim, first note that in round r, p s-sends at least one message (r, WAKEUP) to c. If p s-sends only finitely many messages in round r, then let m be the last message p s-sends to c. By Lemma 11, c receives this message from p infinitely often and this shows the claim. If p s-sends infinitely many messages in round r, then p sends infinitely many messages of round r to c. Moreover, p sends only finitely many messages that are not of round r: this is because (1) in task *retransmit*, p eventually sends only messages of round r, and (2) outside task *retransmit*, p can only send messages of the form ($*$, DECIDE), and such messages are never sent since p never decides. By the link properties, this implies that c receives messages of round r from p infinitely often. This shows the claim.

We now prove that c is an always-up process. In order to obtain a contradiction, suppose that c is an eventually-up process. If c decides after c stabilizes then by Lemma 12 p eventually decides, and this contradicts the assumption that p blocks in round r. So c does not decide after c stabilizes. Then c s-sends a RECOVERED message to p after c stabilizes, and this is the last message c s-sends to p. By Lemma 11, p eventually receives this message and adds c to R_p. So eventually condition $c \notin d'.\mathit{trustlist} \setminus R_p$ in line 68 is true. Therefore, in round r, p's *skip_round* task cannot loop forever in lines 67–68. This contradicts Lemma 9. Hence c is an always-up process. □

Lemma 14. *If the coordinator c of round r is always-up and blocks in round r, then c waits forever at line 25 or 40.*

Proof. Since c is the coordinator of round r and c blocks in round r, c loops forever in lines 22–28 or 36–42, because otherwise q s-sends a DECIDE message to itself (line 43) and then decides (line 11). Since set R_c is finite and c never removes any process from R_c, eventually condition $R_c = \mathit{PrevR}_c$ in lines 28 or 42 is always true. Therefore, c waits forever at line 25 or 40. □

Lemma 15. *Suppose every always-up process proposes. If some good process p decides after p stabilizes, then eventually every good process q decides after q stabilizes.*

Proof. In order to obtain a contradiction, suppose that every always-up process proposes and some good process p decides after p stabilizes, but there is some good process q that does not decide after q stabilizes. Let Q be the set of good processes q such that q does not decide after q stabilizes.

We first claim that Q contains only always-up processes. In order to obtain a contradiction, suppose that $q \in Q$ for some eventually-up process q. Then after q stabilizes, q s-sends a RECOVERED message to all processes, and in particular to process p. This is the last message q s-sends to p. By Lemma 11, p receives RECOVERED messages from q infinitely often. By Lemma 12, q eventually decides after q stabilizes. This contradicts the assumption that $q \in Q$.

So Q contains only always-up processes. By Lemma 8, for every $q \in Q$, q can start only finitely many rounds. Since q proposes, q blocks in some round r_q. Let $r = \max\{r_q \mid q \in Q\}$, and let $q \in Q$ be a process that blocks in round r.

- *Case 1: q is the coordinator of round r.* By Lemma 14, q waits forever at line 25 or 40. Before q waits forever, it s-sends a non-DECIDE message to p (line 24 or 38). By Lemma 11, p receives this message infinitely often. By Lemma 12, q eventually decides after q stabilizes. This contradicts the fact that $q \in Q$.
- *Case 2: q is not the coordinator of round r.* Let $c \neq q$ be the coordinator of round r. By Lemma 13, c is an always-up process and c receives messages of round r from q infinitely often. If c decides after c stabilizes, then by Lemma 12, q decides after q stabilizes and this contradicts the fact that $q \in Q$. So c does not decide after c stabilizes. Since c is always-up, c never decides. By Lemma 10, eventually c starts a round $r' \geq r$. Since $c \in Q$, by the definition of r, we have that $r' \leq r$. Thus $r' = r$ and so c blocks in round r. By Case 1, c eventually decides — a contradiction. □

Henceforth, assume that at most n_b processes are bad, and more than n_b processes are always up.

Lemma 16. *If every always-up process proposes a value, then eventually some always-up process decides.*

Proof. In order to obtain a contradiction, suppose that no always-up process decides. By Lemma 8, every always-up process p can start only finitely many rounds. Since p proposes, p blocks in some round r_p. Let $r = \max\{r_p \mid p$ is always-up$\}$ and let p be an always-up process that blocks in round r.

- *Case 1: p is the coordinator of round r.*
  By Lemma 14, p waits forever at line 25 or 40.

- *Case 1.1: p waits forever at line 25.*
  Let seq be the value of c_seq_p when p waits forever at line 25.
  We first show that for every always-up process q, eventually p receives (r, seq, $estimate_q$, ts_q, ESTIMATE) from q. Process p s-sends (r, seq, NEWROUND) to q (line 24) before p waits forever at line 25. We claim that q receives this message from p and q eventually starts round r. Indeed, if $q = p$, then p receives this message from itself (line 4) and p starts round r by definition. If $q \neq p$, then (r, seq, NEWROUND) is the last message p s-sends to q. By Lemma 11, q eventually receives this message. By Lemma 10, q eventually starts a round $r' \geq r$. By the definition of r, we have that $r' \leq r$. Thus $r' = r$ and so q starts round r.
  Process q cannot receive a NEWESTIMATE message of round r from p, because p waits forever at line 25 and never s-sends NEWESTIMATE messages. So the guard in line 53 is always false. Thus q loops forever in lines 48–53. Since eventually q receives (r, seq, NEWROUND) from p and $seq > 0$ is the largest value of variable c_seq_p in round r, eventually q s-sends (r, seq, $estimate_q$, ts_q, ESTIMATE) to p (line 51) and sets max_seq_q to seq (line 52). We claim that p eventually receives this message from q. Indeed, if $q = p$, then p receives this message from itself (line 4). If $q \neq p$, then (r, seq, $estimate_q$, ts_q, ESTIMATE) is the last message q s-sends to p. By Lemma 11, p eventually receives this message from q. Therefore for every always-up process q, eventually p receives (r, seq, $estimate_q$, ts_q, ESTIMATE) from q.
  Since there are more than n_b processes that are always up, eventually p receives (r, seq, $estimate_q$, ts_q, ESTIMATE) from at least $n_b + 1$ processes. Moreover, for every eventually-up process q, q does not decide after q stabilizes, otherwise by Lemma 15 every always-up process decides. After q stabilizes, q s-sends a RECOVERED message to all (line 75). By Lemma 11, p eventually receives this message from q. When p receives this message from q, p adds q to set R_p (line 9). So eventually R_p contains all eventually-up processes. Since at most n_b processes are bad, eventually the number of always-up processes is at least $n - n_b - |R_p|$. Therefore, eventually p receives (r, seq, $estimate_q$, ts_q, ESTIMATE) from at least $n - n_b - |R_p|$ processes. Hence the guard in line 25 is true forever, and p cannot wait forever at line 25 — a contradiction.
- *Case 1.2: p waits forever at line 40.*
  Let seq be the value of c_seq_p when p waits forever at line 40.
  By an argument analogous to the one in Case 1.1, we can show that: (1) for every always-up process q, p receives (r, seq, ACK) from q; (2) eventually R_p contains all eventually-up processes. Therefore, since at most n_b processes are bad, and more than n_b processes are always up, p receives (r, seq, ACK) from $\max(n_b + 1, n - n_b - |R_p|)$ processes. Hence p cannot wait forever at line 40 — a contradiction.
- *Case 2: p is not the coordinator of round r.*
  Let $c \neq p$ be the coordinator of round r. By Lemma 13, c is an always-up process and c receives messages of round r from p infinitely often. By Lemma 10, c eventually starts a round $r' \geq r$. By the definition of r, we have that $r' \leq r$. Thus $r' = r$ and so c blocks in round r. In Case 1, we showed that the coordinator of round r does not block in round r — a contradiction. □

Corollary 1 (Termination). *If all good processes propose a value, then they all eventually decide.*[13]

Proof. By Lemmata 15 and 16. □

Proof of Theorem 4. Immediate from Lemmata 4 and 7, and Corollary 1. □

7 Solving consensus with stable storage

We now present a consensus algorithm that uses stable storage and $\diamond\mathcal{S}_u$. It requires a majority of good processes and works in systems with lossy links. If the good processes are not a majority, a majority of processes may crash permanently, and so consensus cannot be solved even with $\diamond\mathcal{P}$ and reliable links [3]. Note that requiring a majority of good processes is weaker than requiring $n_a > n_b$, and this is where having stable storage pays off.

The basic structure of the algorithm (given in Fig. 4) is as in [3,4] and consists of rounds of 4 phases each (task *4phases*). In each round r, initially the coordinator c broadcasts a NEWROUND message to announce a new round, and each process sends its current estimate of the decision value — together with a timestamp indicating in which round it was obtained — to c; c waits until it obtains estimates from a majority of processes; it selects one with the largest timestamp and sends it to all processes; every process that receives this new estimate updates its estimate and timestamp accordingly, and sends an acknowledgement to c; when c receives this acknowledgement from a majority of processes, it sends its estimate as the decision to all processes and then it decides. Once a process decides, it stops tasks *4phases* and *retransmit*, and enters a passive state in which, upon receipt of a message, the process responds with the decision value.

A round r can be interrupted by task *skip_round* (which runs in parallel with tasks *coordinator* and *participant*): a process p aborts its execution of round r if (1) it suspects the coordinator c of round r, or (2) it trusts c but detects an increase in the epoch number of c, or (3) it receives a message from a round $r' > r$. When p aborts round r, it jumps to the lowest round $r' > r$ such that p trusts the coordinator of round r' and p has not (yet) received any message with a round number higher than r'.

In each round, a process p accesses the stable storage twice: first to store the current round number, and later to store the new estimate and its corresponding timestamp. Upon recovery, p reads the stable storage to restore its round number, estimate, and timestamp, and then restarts task *4phases* with these values.

Note that in round 1, the coordinator c can simply set its estimate to its *own* proposed value and skip the phase used to select a new estimate (Phase NEWROUND). It is also easy to see that the coordinator does not have to store its round

[13] In fact, it is clear that the following stronger property holds: if all always-up processes propose, then every good process decides after it stabilizes.

```
For every process $p$:
Initialization:
   for all $q \in \Pi \setminus \{p\}$ do $xmitmsg[q] \leftarrow \bot$
To s-send $m$ to $q$:
   if $q \neq p$ then $xmitmsg[q] \leftarrow m$; send $m$ to $q$ else simulate receive $m$ from $p$
Task retransmit:
   repeat forever
      for all $q \in \Pi \setminus \{p\}$ do if $xmitmsg[q] \neq \bot$ then send $xmitmsg[q]$ to $q$
upon propose$(v_p)$:                                   {$p$ proposes $v_p$ by writing it into stable storage}
   $(r_p, estimate_p, ts_p) \leftarrow (1, v_p, 0)$
   fork task {4phases, retransmit}
Task 4phases:
   store $\{r_p\}$; $c_p \leftarrow (r_p \bmod n) + 1$; fork task {skip_round, participant}
   if $p = c_p$ then fork task coordinator
Task coordinator:
   {Phase NEWROUND}
     if $ts_p \neq r_p$ then
       s-send $(r_p$, NEWROUND$)$ to all
       wait until [ received $(r_p, estimate_q, ts_q,$
          ESTIMATE$)$ from $\lceil (n+1)/2 \rceil$ processes ]
       $t \leftarrow$ largest $ts_q$ such that $p$ received
          $(r_p, estimate_q, ts_q,$ ESTIMATE$)$
       $estimate_p \leftarrow$ select one $estimate_q$ such that
          $p$ received $(r_p, estimate_q, t,$ ESTIMATE$)$
       $ts_p \leftarrow r_p$
       store $\{estimate_p, ts_p\}$
   {Phase NEWESTIMATE}
     s-send $(r_p, estimate_p,$ NEWESTIMATE$)$ to all
     wait until [ received $(r_p,$ ACK$)$ from
        $\lceil (n+1)/2 \rceil$ processes ]
     s-send $(estimate_p,$ DECIDE$)$ to all
Task participant:
   {Phase ESTIMATE}
     if $ts_p \neq r_p$ then
       s-send $(r_p, estimate_p, ts_p,$ ESTIMATE$)$ to $c_p$
       wait until [ received $(r_p, estimate_{c_p},$
          NEWESTIMATE$)$ from $c_p$ ]
       if $p \neq c_p$ then
          $(estimate_p, ts_p) \leftarrow (estimate_{c_p}, r_p)$
          store $\{estimate_p, ts_p\}$
   {Phase ACK}
     s-send $(r_p,$ ACK$)$ to $c_p$
Task skip_round:
   $d \leftarrow \mathcal{D}_p$                                                  {query $\diamond\mathcal{S}_u$}
   if $c_p \in d.trustlist$ then
      repeat $d' \leftarrow \mathcal{D}_p$                                       {query $\diamond\mathcal{S}_u$}
      until [ $c_p \notin d'.trustlist$ or $d.epoch[c_p] < d'.epoch[c_p]$ or received some message $(r, \ldots)$ such that $r > r_p$ ]
   terminate task {4phases, participant, coordinator}                  {abort current round}
   repeat $d \leftarrow \mathcal{D}_p$ until $d.trustlist \neq \emptyset$          {query $\diamond\mathcal{S}_u$ to go to a higher round}
   $r_p \leftarrow$ the smallest $r > r_p$ such that $[(r \bmod n) + 1] \in d.trustlist$ and $r \geq \max\{r' \mid p$ received $(r', \ldots)\}$
   fork task 4phases
upon receive $m$ from $q$ do
   if $m = (estimate,$ DECIDE$)$ and decide$(-)$ has not occurred then        {check stable storage about decide}
      decide$(estimate)$                                               {decide is logged into stable storage}
      terminate task {skip_round, 4phases, participant, coordinator, retransmit}
   if $m \neq (-,$ DECIDE$)$ and decide$(estimate)$ has occurred then          {check stable storage about decide}
      send $(estimate,$ DECIDE$)$ to $q$
upon recovery:
   for all $q \in \Pi \setminus \{p\}$ do $xmitmsg[q] \leftarrow \bot$
   if propose$(v_p)$ has occurred and decide$(-)$ has not occurred then   {check stable storage about propose and decide}
      retrieve $\{r_p, estimate_p, ts_p\}$
      if $r_p = \bot$ then $r_p \leftarrow 1$; if $estimate_p = \bot$ then $(estimate_p, ts_p) \leftarrow (v_p, 0)$
      fork task {4phases, retransmit}
```

Fig. 4. Solving Consensus with Stable Storage using $\diamond\mathcal{S}_u$

number in stable storage in this case. We omit these obvious optimizations from the code.

The following regions of code are executed atomically: lines 22–25 and 38–39.

Theorem 5. *The algorithm of Fig. 4 satisfies the Uniform Validity and Uniform Agreement properties of uniform consensus. If a majority of processes are good then it also satisfies the Termination property.*

The proof of this theorem has a similar structure as the proof of Theorem 4, and is given in Appendix C.

8 Performance of the consensus algorithms

8.1 Time and message complexity in nice runs

We analyze the complexity of our algorithms with the optimization in which, in round 1, the coordinator chooses its own estimate and sends it without waiting for estimates from other processes. In most executions of consensus in practice, no process crashes or recovers, no message is lost, the failure detector does not make mistakes, and message delay is bounded by some known δ (including the message processing times). In such "nice" executions, our two algorithms (with and without stable storage) achieve consensus within 3δ: it takes one δ for the coordinator to broadcast NEWESTIMATE messages, one δ for processes to respond with ACKs, and another δ for the coordinator to broadcast DECIDE messages. By adding appropriate delays in the *retransmit* task, so that a message is retransmitted only 2δ time units after it is sent, processes send a total of $4(n-1)$ messages: in the algorithm of Sect. 6, there are $n-1$ messages for each of WAKEUP, NEWESTIMATE, ACK, and DECIDE; in the algorithm of Sect. 7, there are $n-1$ messages for each of the types ESTIMATE, NEWESTIMATE, ACK, and DECIDE.

In contrast, in nice executions the consensus algorithms of [10,7] reach decision within 2δ and with $O(n^2)$ messages. So, compared to our algorithms, they gain one δ in the decision time, at the cost of increasing the message complexity from $O(n)$ to $O(n^2)$. Roughly speaking, this is achieved by distributing the task of collecting ACK's: in our algorithms, the ACK's are sent to the coordinator who counts whether there are enough of them to send a DECIDE to all (this takes 2δ and $O(n)$ messages), while in [10,7] every ACK is broadcast to all processes: each process can then do the counting and deciding by itself (this takes one δ and $O(n^2)$ messages).

8.2 Quiescence

An algorithm is *quiescent* if eventually all processes stop sending messages [1]. It is clear that no consensus algorithm can be quiescent in the presence of unstable processes (each time such a process recovers, it must be sent the decision value, at which point it may crash again and lose this message; this scenario can be repeated infinitely often). If no process is unstable, our consensus algorithms are quiescent despite process crashes and message losses (provided all good processes propose a value).

9 Repeated consensus

In Sects. 6 and 7, and Appendix A, we give algorithms that solve a single instance of consensus. This is appropriate for settings where for each instance of consensus, a distinct set of processes is created to execute it (for example, an application may spawn a new set of processes for each consensus that it wants to do). In other settings, it is necessary for the *same* set of processes to execute repeated (and concurrent) instances of consensus. We now describe how to modify our algorithms to handle this case.

To separate the multiple instances of consensus, each instance must have a unique identifier, and all proposals, decisions, and messages associated with a particular instance of consensus are tagged with the corresponding identifier. This is the only change necessary for the consensus algorithm that uses stable storage (shown in Fig. 4 in Sect. 7).

For the algorithms that do not use stable storage (Fig. 2 in Sect. 6 and Fig. 6 in Appendix A), we can also apply the above modification, except that RECOVERED messages are *not* tagged with instance identifiers (such messages cannot be tagged since a process that recovers has lost all its state). In principle, this modification still works, but in this case the resulting algorithms are not practical because of the following reasons.

A process that recovers from a crash stops participating in *all* subsequent instances of consensus. For a long-lived application this is impractical, since every process is likely to crash and recover at least once during the life of the application, and so eventually no process will remain to run new instances of consensus. Moreover, when a process recovers from a crash, it repeatedly sends a RECOVERED message to get the decision values that it may have "missed" while it was down. When a process receives such a message, it replies with *all* the decision values that it knows — this is also impractical.

To solve the above problems, we now assume that stable storage is available, but each process uses it *only* to store its proposals and decisions (processes do not use it to store any intermediate state, and so, by Theorem 3, solving consensus still requires that $n_a > n_b$). When a process recovers from a crash, it first checks its stable storage to determine which instances of consensus it was executing when it crashed, i.e., the instances for which it proposed a value but did not yet decide. Then, for each such instance I, it sends a RECOVERED message tagged with I, and stops participating in I. With such messages, each process p can now maintain a set R_p^I of processes that it knows to have crashed and recovered *while executing instance* I, and it uses R_p^I instead of R_p. R_p^I is initialized to the empty set when p proposes a value for instance I, and is updated every time p receives a RECOVERED message tagged with I. Finally, if a process receives a RECOVERED message tagged with I and knows the decision value of instance I, then it replies with this decision value.

With these modifications, a process that crashes and recovers can participate in subsequent instances of consensus. Moreover, the algorithm no longer requires that at least $n_b + 1$ processes be always up throughout the lifetime of the system. Instead, it is sufficient that for *each instance* I *of consensus*, at least $n_b + 1$ processes remain up from the time they propose a value for I (to the time they all decide).

10 Transforming $\diamond\mathcal{S}_e$ into $\diamond\mathcal{S}_u$

Figure 5 shows an algorithm to transform $\mathcal{D} \in \diamond\mathcal{S}_e$ into $\mathcal{D}' \in \diamond\mathcal{S}_u$.[14] This transformation works in any asynchronous system with crash and recoveries, provided a majority of processes are good. It does not require any stable storage.

Recall that both $\mathcal{D}$ and $\mathcal{D}'$ require the existence of a good process K such that K is eventually trusted forever by all good processes and K's epoch number at all good processes stops increasing. The difference between $\mathcal{D}$ and $\mathcal{D}'$ is that, while $\mathcal{D}$ allows unstable processes to suspect K or to keep increasing K's epoch number, $\mathcal{D}'$ requires all unstable processes to eventually trust K forever and to stop increasing K's epoch number.

We now explain the main ideas of the algorithm. The output of $\mathcal{D}'$ consists of a trustlist, and epoch numbers for each process on that list. The algorithm maintains the trustlists of $\mathcal{D}'$ as follows. At each process p, initially and every time p recovers, the trustlist of p includes all processes. Process p removes a process from its $\mathcal{D}'$-trustlist only if it finds out that a majority of processes $\mathcal{D}$-suspect this process. With this scheme, if a process K is $\mathcal{D}$-trusted by all the good processes, then K will be $\mathcal{D}'$-trusted by p — even if p is unstable — as required by $\mathcal{D}' \in \diamond\mathcal{S}_u$.

How does p maintain an epoch number for each process in its $\mathcal{D}'$-trustlist? A naive approach would be for p to increment the $\mathcal{D}'$-epoch number of a process q every time p finds out that the $\mathcal{D}$-epoch number of q has increased at a majority of processes. But this does not work, as we now explain. Let u be an unstable process. Suppose that: (a) $n/4$ good processes $\mathcal{D}$-suspect u, (b) $n/4 + 1$ good processes $\mathcal{D}$-trust u while continually increasing its $\mathcal{D}$-epoch number, and (c) all other processes have crashed permanently. In this case: (1) there is no majority that $\mathcal{D}$-suspects u, and (2) there is no majority that $\mathcal{D}$-trusts u and increments its $\mathcal{D}$-epoch number. From (1), a good process p keeps $\mathcal{D}'$-trusting u (see previous paragraph). From (2) and the naive way of generating the $\mathcal{D}'$-epoch numbers, the $\mathcal{D}'$-epoch number of u at p stops changing. So p keeps $\mathcal{D}'$-trusting u and stops increasing its $\mathcal{D}'$-epoch number — a violation of the Completeness property of $\mathcal{D}' \in \diamond\mathcal{S}_u$.

To overcome this problem, p increases the $\mathcal{D}'$-epoch number of a process q every time it finds out that the number of processes that "dislike" q is a majority; a process *dislikes* q if it $\mathcal{D}$-suspects q or it $\mathcal{D}$-trusts q but increases its $\mathcal{D}$-epoch number. This scheme ensures that the $\mathcal{D}'$-epoch number of u keeps on increasing. This also ensures that the $\mathcal{D}'$-epoch number of K stops changing.

In the algorithm, p stores in $latest_p[q]$ the latest output of $\mathcal{D}$ that p received from q (it is initialized to $\bot$).

Theorem 6. *If a majority of processes are good, then the algorithm in Fig. 5 transforms $\diamond\mathcal{S}_e$ into $\diamond\mathcal{S}_u$.*

We now proceed with the proof. Assume that a majority of processes are good. Throughout this proof, let K be some process such that eventually: (1) K is permanently $\mathcal{D}$-trusted by all good processes and (2) the $\mathcal{D}$-epoch number of K at each good process stops changing. The existence of K is guaranteed by the accuracy property of $\mathcal{D} \in \diamond\mathcal{S}_e$.

Lemma 17 (Monotonicity). *At every good process, eventually the $\mathcal{D}'$-epoch numbers are nondecreasing.*

Proof. Clear because, after a good process p stabilizes, for every process q, $epoch_p[q]$ can only be incremented. □

Lemma 18. *For every good process g, eventually g permanently $\mathcal{D}'$-trusts K.*

Proof. Suppose for a contradiction that g $\mathcal{D}'$-suspects K infinitely often. Good processes send messages to g infinitely often, so by the Fair Loss property of links, g receives messages from good processes infinitely often. Thus, g executes line 17 infinitely often as well. When g executes line 17, it $\mathcal{D}'$-suspects K precisely if there is a majority of processes q such that $latest_g[q] \neq \bot$ and $K \notin latest_g[q].trustlist$. Since there is a majority of good processes, every time that g executes line 17 and $\mathcal{D}'$-suspects K, there is some good process q such that $K \notin latest_g[q].trustlist$. Thus, for some good process q, $K \notin latest_g[q].trustlist$ holds infinitely often.

Since q is good, eventually K is permanently $\mathcal{D}$-trusted by q. Then, by the No Creation and Finite Duplication properties of links, eventually g receives no message d_q from q with $K \notin d_q.trustlist$. Since g receives an infinite number of messages from q, eventually $K \in latest_g[q].trustlist$ holds forever — a contradiction. □

Lemma 19. *For every unstable process u, eventually whenever u is up, u $\mathcal{D}'$-trusts K.*

Proof. Suppose for a contradiction that u $\mathcal{D}'$-suspects K infinitely often. Every time u recovers, u sets $\mathcal{D}'_p.trustlist$ to Π, and so since u $\mathcal{D}'$-suspects K infinitely often, it must execute line 17 infinitely often as well. When u executes line 17, it $\mathcal{D}'$-suspects K precisely if there is a majority of processes q such that $latest_u[q] \neq \bot$ and $K \notin latest_u[q].trustlist$. Since there is a majority of good processes, every time that u executes line 17 and $\mathcal{D}'$-suspects K, there is some good process q such that $latest_u[q] \neq \bot$ and $K \notin latest_g[q].trustlist$. Thus, for some good process q, (1) $latest_u[q] \neq \bot$ and (2) $K \notin latest_u[q].trustlist$ hold infinitely often.

When u recovers, it sets $latest_u[q]$ to $\bot$ and, since (1) holds infinitely often, u must set $latest_u[q]$ to a non-$\bot$ value infinitely often. So u receives messages from q infinitely often. Since q is good, eventually K is permanently $\mathcal{D}$-trusted by q. Then, by the No Creation and Finite Duplication properties of links, eventually u receives no message d from q with $K \notin d.trustlist$. Since u receives messages from q infinitely often, eventually $K \in latest_u[q].trustlist$ holds forever. This contradicts the fact that (2) holds infinitely often. □

Lemma 20. *For every good or unstable process p, eventually K's epoch number at p stops changing.*

Proof. Suppose for a contradiction that K's epoch number at p never stops changing. Then p increments $epoch_p[K]$ in line 15 infinitely often. So, $|dislike_p[K]| > n/2$ holds infinitely often, and $dislike_p[K]$ is reset to $\emptyset$ infinitely often. This implies that there exists a majority M of processes such that for every $q \in M$, p infinitely often receives a value d_q from q such that either (1) $K \notin d_q.trustlist$ or (2) ($latest_p[q] \neq \bot$ and $K \in latest_p[q].trustlist$ and $d_q.epoch[K] > latest_p[q].epoch[K]$).

[14] As explained in [3], a transformation algorithm $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ uses failure detector $\mathcal{D}$ to maintain at each process p a variable $\mathcal{D}'_p$ that emulates the output of $\mathcal{D}'$ at p.

```
1  For every process $p$:
2    **Initialization and upon recovery**:
3      $\mathcal{D}'_p.trustlist \leftarrow \Pi$
4      **for all** $q \in \Pi$ **do**
5        $\mathcal{D}'_p.epoch[q] \leftarrow 0$; $epoch_p[q] \leftarrow 0$; $dislike_p[q] \leftarrow \emptyset$; $latest_p[q] \leftarrow \bot$
6    **repeat forever**
7      $d_p \leftarrow \mathcal{D}_p$                                                        {query $\mathcal{D}$}
8      **send** $d_p$ **to** all processes
9    **upon receive** $d_q$ **from** $q$ **do**
10     **for all** $r \in \Pi$ **do**
11       **if** $r \notin d_q.trustlist$ **or** ($latest_p[q] \neq \bot$ **and** $r \in latest_p[q].trustlist$ **and** $d_q.epoch[r] > latest_p[q].epoch[r]$)
             {$p$ determines if $q$ dislikes $r$ (i.e., $q$ does not $\mathcal{D}$-trust $r$ or $q$ increased the $\mathcal{D}$-epoch number of $r$)}
12       **then** $dislike_p[r] \leftarrow dislike_p[r] \cup \{q\}$
13       **if** $|dislike_p[r]| > n/2$ **then**        {if a majority dislikes $r$, $p$ increases the $\mathcal{D}'$-epoch number of $r$}
14         $dislike_p[r] \leftarrow \emptyset$
15         $epoch_p[r] \leftarrow epoch_p[r] + 1$
16     $latest_p[q] \leftarrow d_q$
17     $\mathcal{D}'_p.trustlist \leftarrow \{s : |\{r : latest_p[r] \neq \bot$ **and** $s \notin latest_p[r].trustlist\}| \leq n/2\}$
             {output $\mathcal{D}'$-trust list: $p$ $\mathcal{D}'$-trusts all the processes that are not $\mathcal{D}$-suspected by a majority}
18     **for all** $r \in \mathcal{D}'_p.trustlist$ **do**
19       $\mathcal{D}'_p.epoch[r] \leftarrow epoch_p[r]$                                   {output $\mathcal{D}'$-epoch numbers}
```

Fig. 5. Transforming $\mathcal{D} \in \diamond\mathcal{S}_e$ into $\mathcal{D}' \in \diamond\mathcal{S}_u$

Since a majority of processes is good, there exists a good process $q \in M$. By the No Creation and Finite Duplication properties of the links, q infinitely often sends a value d_q such that either (1) or (2) holds. This implies that either q $\mathcal{D}$-suspects K infinitely often or the $\mathcal{D}$-epoch number of K at q increases infinitely often. Since q is a good process, this contradicts the definition of K. □

Henceforth, let b be a fixed bad process. The Completeness and Monotonicity properties of $\diamond\mathcal{S}_e$ guarantees that for each good process g either (1) eventually g permanently $\mathcal{D}$-suspects b; or (2) eventually the $\mathcal{D}$-epoch number of b at g is nondecreasing and unbounded.

Lemma 21. *For every good processes p and q, line 11 evaluates to true infinitely often for $r = b$.*

Proof. First note that line 11 is executed an infinite number of times since p receives messages from q infinitely often (this follows from the Fair Loss property of links). Suppose that condition $b \notin d_q.trustlist$ does not hold infinitely often. Then eventually $b \in d_q.trustlist$ holds forever. So, eventually all failure detector values that p receives from q contain b in its trustlist. Since p eventually stops crashing, eventually conditions $latest_p[q] \neq \bot$ and $b \in latest_p[q].trustlist$ are always true. Moreover, by the No Creation and Finite Duplication properties of links, q infinitely often sends failure detector values containing b in its trustlist. Therefore q $\mathcal{D}$-trusts b infinitely often. By the Completeness and Monotonicity properties of $\diamond\mathcal{S}_e$, eventually the $\mathcal{D}$-epoch number of b at q is nondecreasing and unbounded. This implies that $d_q.epoch[b] > latest_p[q].epoch[b]$ evaluates to true in line 11 an infinite number of times. □

Lemma 22. *For every good process g, $epoch_g[b]$ is unbounded.*

Proof. Let t_0 be the time after which g does not crash. After t_0, $epoch[b]$ is nondecreasing. For every good process g', g receives messages from g' infinitely often, so it executes line 11 infinitely often as well. So, by Lemma 21, g' is added into set $dislike_g[b]$ an infinite number of times in line 12. By the assumption that a majority of processes are good, $|dislike_g[b]| > n/2$ evaluates to true infinitely often and thus $epoch_g[b]$ grows unboundedly. □

Lemma 23. *For each good process g, either (1) eventually g permanently $\mathcal{D}'$-suspects b; or (2) the $\mathcal{D}'$-epoch number of b at g is unbounded.*

Proof. Let g be any good process and suppose that (1) does not hold. Therefore g $\mathcal{D}'$-trusts b an infinite number of times. Every time g $\mathcal{D}'$-trusts b, it sets the $\mathcal{D}'$-epoch number of b to $epoch_g[b]$. The result now follows from Lemma 22.

Proof of Theorem 6. The Monotonicity property of $\mathcal{D}'$ follows from Lemma 17. Strong Accuracy follows from Lemmata 18, 19 and 20. Completeness follows from Lemma 23. □

A. Solving consensus without stable storage using $\diamond\mathcal{S}_e$

Figure 6 shows the algorithm that solves consensus without stable storage using $\diamond\mathcal{S}_e$ (it is less efficient than the one that uses $\diamond\mathcal{S}'_e$ in Sect. 6). This algorithm always satisfies the Uniform Agreement and Validity properties of uniform consensus, and if the number of processes that are always up is more than n_b, then it also satisfies the Termination property.

In each round k, each process p starts by repeatedly sending its estimate to the current coordinator c (this estimate is called the *k-suggestion of p*). When c receives a k-suggestion, it responds with the *first* k-suggestion that it received. Process p waits for a response from the coordinator until it suspects c or detects an increase in the epoch number of c. If p receives a response from c, it updates its estimate to that value. Then, p sets its $report[k]$ variable to its current estimate —

```
For process $p$:
Initialization:
  $r_p \leftarrow 0$; $R_p \leftarrow \emptyset$
  for all $i \in \mathbf{N}$ do
    $report_p[i] \leftarrow \bot$; $proposal_p[i] \leftarrow \bot$; $coord\_est_p[i] \leftarrow \bot$
upon propose$(v_p)$:                              $\{p$ proposes $v_p$ via an external input containing $v_p\}$
  repeat forever
    $r_p \leftarrow r_p + 1$
    $c_p \leftarrow (r_p \bmod n) + 1$
    repeat send $(r_p, v_p, \text{SUGGESTION})$ to $c_p$
    until [ for some $w$ receive $(r_p, w, \text{ESTIMATE})$ from $c_p$ or suspect $c_p$ or epoch number of $c_p$ increases ]
    if for some $w$ receive $(r_p, w, \text{ESTIMATE})$ from $c_p$ then $v_p \leftarrow w$
    $report_p[r_p] \leftarrow v_p$
    $RV_p[r_p] \leftarrow collect(\text{REPORT})$
    if for some $w$, $RV_p[r_p] = \{w\}$ then $proposal_p[r_p] \leftarrow w$ else $proposal_p[r_p] \leftarrow \lambda$
    $PV_p[r_p] \leftarrow collect(\text{PROPOSAL})$
    if for some $w \neq \lambda$, $w \in PV_p[r_p]$ then $v_p \leftarrow w$
    if for some $w \neq \lambda$, $PV_p[r_p] = \{w\}$ then decide$(w)$
procedure collect(valtype)
  $seq_p \leftarrow 0$
  repeat
    $PrevR_p \leftarrow R_p$; $seq_p \leftarrow seq_p + 1$
    repeat send $(r_p, seq_p, valtype, \text{REQUEST})$ to all
    until [ received messages of the form $(r_p, seq_p, *, valtype)$ from $\max(n_b + 1, n - n_b - |R_p|)$ processes ]
  until $R_p = PrevR_p$
  return({ $v$ : received $(r_p, seq_p, v, valtype)$ })
upon receive RECOVERED from $q$ do
  $R_p \leftarrow R_p \cup \{q\}$
upon receive $(r_q, v_q, \text{SUGGESTION})$ from $q$ do
  if $coord\_est_p[r_q] = \bot$ then $coord\_est_p[r_q] \leftarrow v_q$
  send $(r_q, coord\_est_p[r_q], \text{ESTIMATE})$ to $q$
upon receive $(r_q, seq_q, \text{REPORT}, \text{REQUEST})$ from $q$ do
  if $report_p[r_q] \neq \bot$ then send $(r_q, seq_q, report_p[r_q], \text{REPORT})$ to $q$
upon receive $(r_q, seq_q, \text{PROPOSAL}, \text{REQUEST})$ from $q$ do
  if $proposal_p[r_q] \neq \bot$ then send $(r_q, seq_q, proposal_p[r_q], \text{PROPOSAL})$ to $q$
upon recovery:
  for all $i \in \mathbf{N}$ do
    $report_p[i] \leftarrow \bot$; $proposal_p[i] \leftarrow \bot$; $coord\_est_p[i] \leftarrow \bot$
  repeat forever
    send RECOVERED to all
```

Fig. 6. Solving Consensus without Stable Storage using $\diamond\mathcal{S}_e$

this is the *k-report of p*. After this, p collects the k-reports of other processes (the collect procedure is explained below). If all the collected k-reports are for the *same* value, then p sets its $proposal[k]$ variable to that value; otherwise, p sets it to the special value "λ" (which cannot be one of the proposed values) — this is the *k-proposal of p*. Then, p collects the k-proposals of other processes. If *some* collected k-proposal w is different from λ, then p sets its estimate to w (we will show that it cannot collect two distinct k-proposals different from λ). Moreover, if *all* collected k-proposals are for w, p decides w.

When a process recovers from a crash, it stops participating in the algorithm except that: (1) it periodically broadcasts a RECOVERED message, and (2) if asked to act as the coordinator for some round r (by receiving an r-suggestion) it will do so. When a process p receives a RECOVERED message from some process q, it adds q to a set R_p of processes known to have recovered.

To collect k-reports, a process p invokes procedure *collect*(REPORT). In this procedure, p repeatedly sends requests for the k-reports of other processes; when a process receives such a request, it sends back its k-report if it is different from $\bot$. After p has received k-reports from $\max(n_b + 1, n - n_b - |R_p|)$ processes, it checks whether during the collection of k-reports it detected the recovery of a process that never recovered before ($R_p \neq PrevR_p$). If so, p restarts the collection of k-reports from scratch; else, p returns from procedure *collect*(REPORT). Process p collects k-proposals in a similar way.

To illustrate the main ideas of the algorithm, we made two simplifications. First, we did not require that all *good* processes decide: in fact, this algorithm only guarantees that all *always-up* processes eventually decide. Second, we assumed that links satisfy the following *Per-Message Fair Loss* property (instead of the *Fair Loss* property of Sect. 2.5): if a process p sends a message m to a good process q an infinite number of times, then q receives m from p an infinite number of times.[15] We later remove these two simplifications by modifying the algorithm so that: (1) all good processes eventually decide (and eventually stop executing the algorithm), and (2) the algorithm works with links that satisfy the Fair Loss property of Sect. 2.5.

Theorem 7. *The algorithm of Fig. 6 satisfies the Uniform Validity and Uniform Agreement properties of uniform consensus. Moreover, suppose that at most n_b processes are bad, more than n_b processes are always up, and links satisfy the Per-Message Fair Loss property. If all always-up processes propose a value, then they all eventually decide.*

The proof follows.

Definition 7. *We say that* p is in round r at time t *if p does not crash by time t and the value of variable r_p at time t is r. A process* p starts round r *when p sets variable r_p to r in line 7. Process* p reaches the end of round r *if p completes the execution of the loop in lines 7–17 in round r.*

Definition 8. *We say that* p k-reports v *if it sets $report_p[k]$ to v in line 12 in round k. Similarly, we say that* p k-proposes v *if it sets $proposal_p[k]$ to v in line 14.*

Definition 9. *We say that* p completes the collection of k-reports *if it returns from the invocation of collect(*REPORT*) and sets $RV_p[k]$ to the return value in line 13 in round k. Similarly, we say that* p completes the collection of k-proposals *if it returns from the invocation of collect(*PROPOSAL*) and sets $PV_p[k]$ to the return value in line 15 in round k.*

Lemma 24 (Uniform Validity). *If a process decides v then some process previously proposed v.*

Proof. A simple but tedious induction shows that the variable v_p of any process p is always set to some value that was previously proposed by some process. Moreover, clearly the decision value is the value of variable v_p of some process p at some time.

Lemma 25. *For any processes p and q that complete the collection of k-reports $RV_p[k] \cap RV_q[k] \neq \emptyset$.*

Proof. For any process p that completes the collection of k-reports, p invokes *collect*(REPORT) and returns from this invocation. During this invocation, consider the time when p executes line 24 for the last time, and at this time let:

- s_p be the value of seq_p;
- P_p be the subset of processes from which p has received $(k, s_p, *, \text{REPORT})$;
- $\overline{R}_p$, the value of set R_p.

Clearly, to show that $RV_p[k] \cap RV_q[k] \neq \emptyset$, it is sufficient to show that $P_p \cap P_q \neq \emptyset$. We first claim that (1) processes in $\overline{R}_p$ crash before p starts sending (k, s_p, REPORT, REQUEST) to any process in round k (line 22). Indeed, when p executes line 24 for the last time during its invocation of *collect*(REPORT) in round k, we have that $\overline{R}_p = R_p$ (by the definition of $\overline{R}_p$) and $R_p = PrevR_p$ (by the condition in line 24). Therefore, $PrevR_p = \overline{R}_p$. All processes in $PrevR_p$ crash before p starts sending (k, s_p, REPORT, REQUEST), and so the claim follows.

Note that (2) $|P_p| \geq n_b + 1$ (this is due to the guard in line 23). We now show that $P_p \cap \overline{R}_p = \emptyset$. Let $p' \in \overline{R}_p$. By the previous claim, p' crashes before p starts sending (k, s_p, REPORT, REQUEST). This happens before any process sends (k, s_p, $*$, REPORT) to p. So p' crashes before any process sends (k, s_p, $*$, REPORT) to p. Since after a process crashes (and recovers) it does not send REPORT messages, it follows that $p' \notin P_p$. Thus, $P_p \cap \overline{R}_p = \emptyset$.

So, $|P_p \cup \overline{R}_p| = |P_p| + |\overline{R}_p|$. By the threshold in line 23, we have $|P_p| \geq \max(n_b+1, n-n_b-|\overline{R}_p|)$, and thus (3) $|P_p \cup \overline{R}_p| \geq n - n_b$.

By the same argument, we have (4) $|P_q| \geq n_b + 1$ and (5) $|P_q \cup \overline{R}_q| \geq n - n_b$.

Now suppose, in order to obtain a contradiction, that $P_p \cap P_q = \emptyset$. By (3) and (4), we have $P_q \cap (P_p \cup \overline{R}_p) \neq \emptyset$. Since $P_p \cap P_q = \emptyset$, we have $P_q \cap \overline{R}_p \neq \emptyset$. Let $p' \in P_q \cap \overline{R}_p$. Clearly, q starts sending (k, s_q, REPORT, REQUEST) to processes before p' receives such a message, which happens before p' sends a message of the form (k, s_q, $*$, REPORT) to q (p' sends such message because $p' \in P_q$), which happens before p' crashes (since after a process crashes and recovers, it does not send REPORT messages), which happens before p starts sending (k, s_p, REPORT, REQUEST) to processes (this follows from the fact that $p' \in \overline{R}_p$ and Claim (1)). From all this, we conclude that q starts sending (k, s_q, REPORT, REQUEST) before p starts sending (k, s_p, REPORT, REQUEST).

By (2) and (5), we have $P_p \cap (P_q \cup \overline{R}_q) \neq \emptyset$. By an argument analogous to the one above, we can conclude that p starts sending (k, s_p, REPORT, REQUEST) before q starts sending (k, s_q, REPORT, REQUEST). This is a contradiction. Hence, we conclude that $P_p \cap P_q \neq \emptyset$.

Lemma 26. *If p and q k-propose $v \neq \lambda$ and $v' \neq \lambda$, respectively, then $v = v'$.*

Proof. If p k-proposes $v \neq \lambda$, then p sets $proposal_p[k]$ to v in line 14. Thus $RV_p[k] = \{v\}$. Similarly we have $RV_q[k] = \{v'\}$. By Lemma 25, $RV_p[k] \cap RV_q[k] \neq \emptyset$. Therefore, $v = v'$.

Lemma 27. *If p completes the collection of k-proposals, then $PV_p[k]$ contains at most one value different from λ.*

Proof. In order to obtain a contradiction, suppose that this is not true, i.e., there exist $v \neq \lambda$ and $v' \neq \lambda$ such that $v \neq v'$ and $v, v' \in PV_p[k]$. Every value in $PV_p[k]$ is k-proposed by some process, so there exist processes q and q' that k-propose v and v', respectively. By Lemma 26, $v = v'$ — a contradiction.

Lemma 28. *For any processes p and q that complete the collection of k-proposals, $PV_p[k] \cap PV_q[k] \neq \emptyset$.*

[15] The Fair Loss and Per-Message Fair Loss properties of links are called Weak Loss Limitation and Strong Loss Limitation, respectively, in [8].

Proof. This proof is similar to the proof of Lemma 25.

Lemma 29. *If in round k some process p decides v, then all processes q that reach the end of round k set variable v_q to v in line 16. Moreover, if q decides v' in round k then $v = v'$.*

Proof. Since in round k p decides v, then $v \neq \lambda$ and $PV_p[k] = \{v\}$. For every process q that reaches the end of round k, q completes the collection of k-proposals, and thus by Lemma 28, $v \in PV_q[k]$. By Lemma 27, v is the only value in $PV_q[k]$ different from λ, so q sets variable v_q to v in line 16. Moreover, since $v \in PV_q[k]$, if q decides v' then $v = v'$.

Lemma 30. *If all processes p that reach the end of round k set variable v_p to v in line 16, then all processes that $(k+1)$-report a value $(k+1)$-report v.*

Proof. This is clear from the fact that every value $(k+1)$-reported is the value of variable v_p at the end of round k for some process p.

Lemma 31. *If all processes that k-report a value k-report the same value v, then all processes p that reach the end of round k set variable v_p to v in line 16 and decide v.*

Proof. First note that processes cannot k-report λ, because no process p can set its variable v_p to λ at any time. If all processes that k-report a value k-report the same value v, then for all processes p that complete the collection of k-reports, $RV_p[k] = \{v\}$. Thus all processes that k-propose a value k-propose the same value v. Therefore, for all processes p that complete the collection of k-proposals, $PV_p[k] = \{v\}$. Since $v \neq \lambda$, all processes p that reach the end of round k set variable v_p to v in line 16 and decide v.

Lemma 32 (Uniform Agreement). *No two processes decide differently.*

Proof. Suppose process p decides v in round k and process q decides v' in round k'. We show that $v = v'$.

Assume without loss of generality that $k \leq k'$. If $k = k'$, then $v' = v$ by Lemma 29. Now suppose $k < k'$. By Lemma 29, all processes p' that reach the end of round k set variable $v_{p'}$ to v in line 16. By Lemma 30, all processes that $(k+1)$-report a value $(k+1)$-report v. By Lemma 31, all processes p' that reach the end of round $k+1$ set variable $v_{p'}$ to v in line 16 and decide v. By repeatedly applying Lemmata 30 and 31, we conclude that all processes that reach the end of round k' decide v. Since q reaches the end of round k', it decides v in round k', and so $v = v'$.

Henceforth assume that at most n_b processes are bad, more than n_b processes are always up, and links satisfy the Per-Message Fair Loss property.

Lemma 33. *If an always-up process p starts a round k, then eventually it k-reports a value.*

Proof. In order to obtain a contradiction, suppose that p never k-reports any value. Then p loops forever in lines 9–10 in round k. Let c be the coordinator of round k. If c is a bad process, then according to the Monotonicity and Completeness property of $\diamond\mathcal{S}_e$, either eventually p permanently suspects c or the epoch number of c at p is nondecreasing and unbounded. Thus eventually the guard in line 10 is true and p does not loop forever in lines 9–10. So c is a good process. Process p sends $(k, v_p, \text{SUGGESTION})$ to c infinitely often (line 9). By the Per-Message Fair Loss property, c receives this message from p infinitely often. Since c is a good process, there is a time t after which c does not crash. After time t, every time c receives $(k, v_p, \text{SUGGESTION})$ from p, c sends the same message $(k, w, \text{ESTIMATE})$ to p. So c sends $(k, w, \text{ESTIMATE})$ to p infinitely often. By the Per-Message Fair Loss property, p eventually receives this message. Therefore, p does not loop forever in lines 9–10 — a contradiction.

Lemma 34. *If all always-up processes k-report a value, then eventually they all k-propose a value.*

Proof. In order to obtain an contradiction, suppose that all always-up processes k-report a value, but there is an always-up process p that never k-proposes any value. So p never returns from the invocation of *collect*(REPORT) in round k. Process p loops forever either in lines 20–24 or 22–23. Since set R_p is finite and p never removes any process from R_p, eventually condition $R_p = \mathit{PrevR}_p$ in lines 24 is always true. Therefore, p loops forever in lines 22–23. Thus for some value s_p, p sends $(k, s_p, \text{REPORT}, \text{REQUEST})$ to all processes infinitely often.

For every always-up process q, by the Per-Message Fair Loss property, q receives $(k, s_p, \text{REPORT}, \text{REQUEST})$ from p infinitely often. Since q k-reports a value, there is a time t after which $\mathit{report}_q[k] = w$ for some $w \neq \bot$. So after time t, every time q receives $(k, s_p, \text{REPORT}, \text{REQUEST})$ from p, q sends $(k, s_p, w, \text{REPORT})$ to p (line 32). Thus q sends $(k, s_p, w, \text{REPORT})$ to p infinitely often. By the Per-Message Fair Loss property, eventually p receives $(k, s_p, w, \text{REPORT})$ from q. Therefore eventually p receives messages of the form $(k, s_p, *, \text{REPORT})$ from all always-up processes. Since more than n_b processes are always up, eventually p receives messages of the form $(k, s_p, *, \text{REPORT})$ from at least $n_b + 1$ processes.

For every eventually-up process q, it is clear that eventually p receives a RECOVERED message from q, since after q's last recovery q sends RECOVERED messages to all processes infinitely often. Therefore, eventually R_p contains all eventually-up processes. Since there are at most n_b bad processes, eventually the number of always-up processes is at least $n - n_b - |R_p|$. Therefore eventually p receives messages of the form $(k, s_p, *, \text{REPORT})$ from at least $n - n_b - |R_p|$ processes.

Hence, eventually p receives messages of the form $(k, s_p, *, \text{REPORT})$ from $\max(n_b + 1, n - n_b - |R_p|)$ processes, so the guard in line 23 is true. Therefore process p does not loop forever in lines 22–23 — a contradiction.

Lemma 35. *If all always-up processes k-propose a value then eventually they all reach the end of round k.*

Proof. Similar to the proof of Lemma 34.

Corollary 2. *If all always-up processes propose, then for every $k \in \{1, 2, 3, \ldots\}$, eventually they all reach the end of round k.*

Proof. If all always-up processes propose, they all start round 1. Lemmata 33, 34 and 35 show that if all always-up processes start a round r then eventually they all reach the end of round r; thus, they all start round $r+1$. The proof follows by induction.

Lemma 36. *There exists a round k such that all processes that k-report a value k-report the same value.*

Proof. Choose a time T such that (1) all processes that are not always-up have crashed at least once by time T, (2) all good processes remain up forever after time T, and (3) for some good process c, for every good process g, after time T, g permanently trusts c and the epoch number of c at g stops changing (we can find such process c by the Accuracy property of $\diamond\mathcal{S}_e$). Choose a round k such that no process starts round k by time T, and c is the coordinator of round k.

Let p be a process that k-reports a value. Then p eventually exits the loop in lines 9–10. Moreover, by definition of k, p starts round k after time T. Only always-up processes can start a round after time T, because all other processes crashed at least once by time T and, after they crash, they never start any round. Thus, p is an always-up process, and so in round k, p never suspects c and the epoch number of c at p never increases. Thus, p can only exit the loop in lines 9–10 by receiving $(k, w,$ ESTIMATE$)$ from c, for some $w \neq \bot$. Since p eventually exits this loop, it receives $(k, w,$ ESTIMATE$)$ from c. Therefore, there is a time at which $coord\_est_c[k] = w$. Note that c never receives any message of the form $(k, *,$ SUGGESTION$)$ by time T, because no process starts round k by time T. Therefore, the value of $coord\_est_c[k]$ is $\bot$ before or at time T. Thus, c sets $coord\_est_c[k]$ to w after time T. Since c does not crash after time T, once c sets $coord\_est_c[k]$ to w, it never changes this variable again. This implies that every process that k-reports a value receives $(k, w,$ ESTIMATE$)$ from c, and then k-reports w.

Lemma 37. *If all always-up processes propose a value then they all eventually decide.*

Proof. Suppose that all always-up processes propose a value. By Lemma 36, there exists a round k such that all processes that k-report a value k-report the same value. By Corollary 2, all always-up processes reach the end of round k. By Lemma 31, all always-up processes decide in round k.

Proof of Theorem 7. Immediate from Lemmata 24, 32, and 37. □

We now explain how to remove the two limitations that we mentioned at the beginning of this section. The first one is that the algorithm in Fig. 6 does not guarantee that eventually-up processes decide; moreover processes never stop executing rounds. To fix these problems, we modify the algorithm as follows. Once a process p decides, it stops executing the algorithm. Then, every time that p receives any message it replies with the decision value. When a process receives the decision value, it decides. With this modification, all good processes decide and all processes eventually stop executing rounds.

The second limitation is that the algorithm does not work with the Fair Loss property of Sect. 2.5. We first explain why, and then we modify the algorithm to fix this problem.

There are two types of messages in the algorithm: *active* messages, i.e., those that are actively sent by processes (SUGGESTION, REQUEST and RECOVERED messages), and *passive* messages, which are sent in response to an active message (ESTIMATE, REPORT, PROPOSAL and "decide" messages). In the algorithm, a process p proceeds by sending an active message to other processes, until it gets responses; then p sends a different active message, and so on. The problem arises when p repeatedly sends an active message to q, while q repeatedly sends another active message to p. Every time p receives the active message from q, p replies with a passive message, and vice-versa. Thus, p repeatedly sends both an active and a passive message to q, and vice-versa. With the Fair Loss property, it is possible that all the active messages are received and all the passive ones are lost. Thus, p and q never receive a reply from each other.

To fix this problem, we modify the algorithm as follows. For all p and q, process p now keeps a copy of the last message of each type (active or passive) that it wants to send to q. Every time p sends an active or passive message to q in the original algorithm, in the modified algorithm it actually sends a tuple consisting of *both* the last active and the last passive messages to q. When q receives such a tuple, it processes both components separately (as if q had received both messages separately in the original algorithm). With this modification, the algorithm will work with the Fair Loss property.

From the above, we have:

Theorem 8. *Assume that at most n_b processes are bad and more than n_b processes are always up. Uniform consensus can be solved without stable storage using $\diamond\mathcal{S}_e$.*

B. Implementation of $\diamond\mathcal{S}_e$ and $\diamond\mathcal{S}'_e$ in partially synchronous systems

We show how to implement $\diamond\mathcal{S}_e$ and $\diamond\mathcal{S}'_e$ in the models of partial synchrony of [5,3] (extended to systems with crashes and recoveries). [5] considers two models of partial synchrony. Roughly speaking, the first model, denoted $\mathcal{M}_1$ here, stipulates that in every execution there are bounds on process speeds and on message transmission times, but these bounds are not known. In the second model, denoted $\mathcal{M}_2$, these bounds are known, but they hold only after some unknown time (called *GST* for *Global Stabilization Time*). [3] defines a weaker model of partial synchrony, denoted $\mathcal{M}_3$, in which bounds exist but they are not known *and* they hold only after some unknown GST. In $\mathcal{M}_1$ links do not lose messages, and in $\mathcal{M}_2$ and $\mathcal{M}_3$ links can only lose messages sent before the GST. Note that every system that conforms to $\mathcal{M}_1$ or $\mathcal{M}_2$ also conforms to $\mathcal{M}_3$.

All the above models assume that process crashes are permanent. A natural extension of $\mathcal{M}_3$ to systems with crashes and recoveries, which we also denote $\mathcal{M}_3$, is as follows: after some (unknown) GST, all the good processes are up forever, and there are bounds on process speeds and on message transmission times. In particular, all the messages sent to good processes after the GST, including those sent by unstable processes, are received within the (unknown) bound. Messages sent to bad processes may be lost. Henceforth, $\mathcal{M}_3$ denotes this extended model.

Figure 7 shows an implementation of $\diamond\mathcal{S}_e$ (and also of $\diamond\mathcal{S}'_e$) in $\mathcal{M}_3$. The algorithm is similar to one given in [3]. To measure elapsed time, each process p maintains a local clock, say, by counting the number of steps that it takes. After each recovery, each process p first sends an I-RECOVERED message to all processes; then it periodically sends an I-AM-ALIVE message. If p does not receive an I-AM-ALIVE message from

```
For process $p$:
  **Initialization and upon recovery**:
    $\mathcal{D}_p.trustlist \leftarrow \Pi$; $trustlist_p \leftarrow \Pi$
    **for all** $q \in \Pi$ **do** $\mathcal{D}_p.epoch[q] \leftarrow 0$; $epoch_p[q] \leftarrow 0$; $\Delta_p[q] \leftarrow$ default time-out interval
    **send** I-RECOVERED **to** all processes
  **repeat forever**
    **send** I-AM-ALIVE **to** all processes
    **for all** $q \in \Pi$ **do**
      **if** $q \in trustlist_p$ and $p$ did not **receive** I-AM-ALIVE from $q$ during the last $\Delta_p[q]$ ticks of $p$'s clock **then**
        $trustlist_p \leftarrow trustlist_p \setminus \{q\}$                                  {suspect $q$}
    $\mathcal{D}_p.trustlist \leftarrow trustlist_p$                                   {update the failure detector output}
    **for all** $q \in \mathcal{D}_p.trustlist$ **do** $\mathcal{D}_p.epoch[q] \leftarrow epoch_p[q]$
  **upon receive** I-AM-ALIVE **from** $q$ **do**
    **if** $q \notin trustlist_p$ **then**
      $trustlist_p \leftarrow trustlist_p \cup \{q\}$                                      {trust $q$}
      $\Delta_p[q] \leftarrow \Delta_p[q] + 1$                                             {increase timeout}
  **upon receive** I-RECOVERED **from** $q$ **do**
    $epoch_p[q] \leftarrow epoch_p[q] + 1$
```

Fig. 7. Implementing $\diamond\mathcal{S}_e$ and $\diamond\mathcal{S}'_e$ in $\mathcal{M}_3$

some process q for $\Delta_p[q]$ time units on its clock, p removes q from its list of trusted processes. When p receives I-AM-ALIVE from some process q, it checks if it currently suspects q. If so, p knows that its previous time-out on q was premature and so p adds q to its list of trusted processes and increases its time-out period $\Delta_p[q]$. When p receives I-RECOVERED from some process q, it increments the epoch number of q. Note that this implementation does not use any stable storage.

Following [3], it is easy to see that when this algorithm is executed in $\mathcal{M}_3$, there is a time after which every good process trusts every good process and suspects every eventually-down process. It is also easy to see that at every good process, eventually the epoch numbers are nondecreasing (this occurs after the process stops crashing). Moreover, good processes send I-RECOVERED messages only a finite number of times, so that the epoch numbers of each good process at every good process eventually stop changing. It remains to show that for every unstable process u and every good process g, either eventually g permanently suspects u or u's epoch number at g is unbounded. Indeed, if g does not permanently suspect u, then it trusts u infinitely often; in this case, g receives I-AM-ALIVE messages from u infinitely often. So u sends I-AM-ALIVE messages to g infinitely often. Note that after each recovery, u always sends I-RECOVERED message before sending I-AM-ALIVE messages. Therefore, u sends I-RECOVERED messages infinitely often. Thus, g receives I-RECOVERED messages from u infinitely often and so g increments u's epoch number infinitely often.

Hence we have:

Theorem 9. *In any partially synchronous system that conforms to $\mathcal{M}_3$, the algorithm in Fig. 7 guarantees that (1) at every good process, eventually the epoch numbers are nondecreasing, (2) for every bad process b and every good process g, either eventually g permanently suspects b or b's epoch number at g is unbounded, and (3) for every good process g, eventually g is permanently trusted by every good process, and g's epoch number at every good process stops changing.*

Corollary 3. *In any partially synchronous system that conforms to $\mathcal{M}_3$, the algorithm in Fig. 7 implements $\diamond\mathcal{S}_e$ and $\diamond\mathcal{S}'_e$.*

Note that the algorithm does *not* implement $\diamond\mathcal{S}_u$ in $\mathcal{M}_3$. This is because an unstable process u resets its timeouts to a default value infinitely often, and if this value is smaller than the (unknown) bound on message delays, then u may suspect *every* process infinitely often — a violation of the strong accuracy property of $\diamond\mathcal{S}_u$. In Sect. 10, however, we show how to transform any implementation of $\diamond\mathcal{S}_e$ into $\diamond\mathcal{S}_u$ (this transformation does *not* rely on partial synchrony assumptions).

C. Proof of Theorem 5

Theorem 5 *The algorithm of Fig. 4 satisfies the Uniform Validity and Uniform Agreement properties of uniform consensus. If a majority of processes are good then it also satisfies the Termination property.*

The proof follows.

Definition 10. *We say that* p is in round r at time t *if the value of variable r_p in stable storage at time t is r. A process* p starts round r *when p stores r as the value of r_p for the first time in line 12. We say that* p updates $estimate_p$ to est *when p stores est as the value of $estimate_p$ (in line 25 or 39). Similarly, we say that* p updates ts_p to t *when p stores t as the value of ts_p (in line 25 or 39).*

Lemma 38 (Uniform Validity). *If a process decides v then some process previously proposed v.*

Proof. Trivial. □

Lemma 39. *A process can update $estimate_p$ and ts_p at most once in each round.*

Proof. Let r be a round and p be a process. In round r, if p is the coordinator of round r then p can only update $estimate_p$

and ts_p in line 25; else, p can only update $estimate_p$ and ts_p in line 39. When p updates $estimate_p$ and ts_p, it updates ts_p to r. After it does so, it can not execute lines 25 and 39 in round r again (even if it crashes and later recovers) because of the guard in lines 16 and 33, respectively. □

Lemma 40. *Let c be the coordinator of some round r. (1) In round r, if c starts Phase* NEWESTIMATE *with $estimate_c = est$, then c updates $estimate_c$ to est; and (2) in some round $r' > r$, if some process p* s-sends *(r', est, r,* ESTIMATE*) in line 34, then in round r, c updates $estimate_c$ to est.*

Proof. To prove (1), assume that in round r, c starts Phase NEWESTIMATE with $estimate_c = est$. Clearly, before c starts Phase NEWESTIMATE, it updates $estimate_c$ to some value est'. By Lemma 39, c updates $estimate_c$ at most once in round r. Therefore $est = est'$. This shows (1).

To prove (2), assume that in some round $r' > r$ some process p s-sends (r', est, r, ESTIMATE) in line 34. We first claim that in round r, p updates $estimate_p$ to est. Indeed, since $ts_p = r$ when p executes line 34 in round r', p must have executed line 24 or 38 in round r to set ts_p to r, and then stored ts_p in line 25 or 39 in round r. Let est' be the value of $estimate_p$ that p stores in line 25 or 39 in round r. We need to show that $est' = est$. Indeed, it is clear that when p executes line 34 in round r', the values of $estimate_p$ and ts_p in stable storage are est and r, respectively (this is because every time p changes $estimate_p$ or ts_p, it stores its new value in stable storage — see lines 25 and 39). Moreover, from the structure of the algorithm, the value of ts_p in stable storage is nondecreasing, so that after p stores ts_p in round r, its value in stable storage does not change until round r'. Note that $estimate_p$ and ts_p are always updated together. So after p stores $estimate_p$ in round r, the value of $estimate_p$ in stable storage also does not change until round r'. So $est' = est$, and this shows the claim.

Now there are two cases. If $p = c$ (i.e., p is the coordinator of round r), then part (2) follows immediately from the claim. If $p \neq c$, then p does not execute line 25 in round r, and so by the claim p stores est as the value of $estimate_p$ in line 39 in round r. Thus p must have received (r, est, NEWESTIMATE) from c, which implies that c must have s-sent this message to p in line 27 in round r. By part (1), c updates $estimate_c$ to est in round r. □

Lemma 41. *Suppose that the coordinator c of round r* s-sends *(est,* DECIDE*) in line 30. In every round $r' \geq r$, if the coordinator c' updates $estimate_{c'}$ to some value est' then $est = est'$.*

Proof. We prove this lemma by induction on the round number r'. For the base case ($r' = r$), note that if c s-sends (est, DECIDE) in line 30, then c starts Phase NEWESTIMATE with $estimate_c = est$. The base case now follows directly from Lemmata 39 and 40 (1).

Now assume that the lemma holds for all $r', r \leq r' < k$. Let c' be the coordinator of round k. We show that the lemma holds for $r' = k$.

Suppose that in round k, c' updates $estimate_{c'}$ to some value est'. Since c' is the coordinator of round k, this update can only happen in line 25. Then c' received messages of the form ($k, *, *,$ ESTIMATE) from $\lceil (n+1)/2 \rceil$ processes in the wait statement in line 18. Since c executes line 30 in round r, c receives (r, ACK) from $\lceil (n+1)/2 \rceil$ processes. Thus, there is some process p such that (1) in round r, c receives (r, ACK) from p, and (2) in round k, for some est'' and ts'', c' receives (k, est'', ts'', ESTIMATE) from p in the wait statement in line 18. By (1), p s-sends (r, ACK) to c in round r. By (2), p s-sends (k, est'', ts'', ESTIMATE) to c' in round k. Before doing that, p starts round k. After p starts round k, p never starts a round lower than k. This implies that p s-sends (k, est'', ts'', ESTIMATE) to c' in round k after p s-sends (r, ACK) to c in round r. Before s-sending (r, ACK) to c in round r, p updates ts_p to r (line 25 or 39). Since the value of ts_p in stable storage is non-decreasing, we must have $ts'' \geq r$. It is easy to see that no process ever s-sends a message of the form ($k, *, ts$, ESTIMATE) with $ts \geq k$. So, the value t that c' selects in line 20 in round k is such that $r \leq t < k$. Let q be the process whose estimate value est' is selected in line 22 in round k. Then in round k, q s-sends (k, est', t, ESTIMATE). By Lemma 40 (2), the coordinator c'' of round t updated $estimate_{c''}$ to est'. By the induction hypothesis, we have $est' = est$. □

Lemma 42. *If processes c and c'* s-send *(est,* DECIDE*) and (est',* DECIDE*) in line 30 in rounds r and r', respectively, then $est = est'$.*

Proof. Assume without loss of generality that $r' \geq r$. Since line 30 is executed only by the coordinator, c and c' are the coordinators of rounds r and r', respectively. Since c' s-sends (est', DECIDE) in line 30 in round r', c' starts Phase NEWESTIMATE with $estimate_{c'} = est'$. By Lemma 40 (1), c' updates $estimate_{c'}$ to est'. By Lemma 41, $est = est'$. □

Lemma 43 (Uniform Agreement). *No two processes decide differently.*

Proof. Suppose that processes p and p' decide on values est and est', respectively. Process p decides est in line 53 after receiving message (est, DECIDE). By a simple induction, some process must have s-sent message (est, DECIDE) in line 30. Similarly, process p' decides est' in line 53, and so some process must have s-sent message (est', DECIDE) in line 30. By Lemma 42, $est = est'$. □

Lemma 44. *A process can start only finitely many rounds.*

Proof. In order to obtain a contradiction, suppose that there are processes that start infinitely many rounds. Let P be the set of all such processes. Clearly, P contains only good or unstable processes. For any process $p \in P$ and any round $r \geq 1$, p eventually starts a round higher than r. Let r_p^+ be the lowest round higher than r that p starts and let r_p^- be the highest round lower than or equal to r that p starts. Then $1 \leq r_p^- \leq r < r_p^+$.

By the Strong Accuracy property of $\diamond\mathcal{S}_u$, we can find a time T and a good process K such that after T, K is never suspected by any good or unstable process and the epoch number of K at every good or unstable process stops changing.

Let r be a round such that (1) K is the coordinator of round r, and (2) no process in $\Pi \setminus P$ starts a round higher than r, and (3) for every $p \in P$, p starts round r_p^- after time T. Such round clearly exists because processes in $\Pi \setminus P$ start only finitely many rounds and processes in P start infinitely many rounds.

Let p be the first process to start a round higher than r. By (2), $p \in P$ and by the definition of r_p^- and r_p^+, p selects round r_p^+ when it executes line 49 in round r_p^-. This implies that $r_p^- = r$: indeed, if $r_p^- < r$ then p does not select round r_p^+ in line 49; instead, it selects a round number that is at most r since (a) p trusts the coordinator K of round r (by (3) and the definitions of T and K), and (b) p does not receive any messages of a round higher than r (since p is the first process to start a round higher than r). So $r_p^- = r$. By (3), p starts round r after time T. By (1) and the definition of T and K, while p is in round r, condition $K \in d.trustlist$ in line 44 evaluates to true and condition $(K \notin d'.trustlist$ **or** $d.epoch[K] < d'.epoch[K])$ in line 46 always evaluates to false. Since p starts a round higher than r, it does not loop forever in lines 45–46. So p eventually receives a message of a round higher than r while in round r. This contradicts the fact that p is the first process to start a round higher than r. □

Definition 11. *We say that a good process p blocks in round r if p starts round r but p does not start a higher round, and p never decides.*

Lemma 45. *If a good process p blocks in round r, then in this round its skip_round task loops forever in lines 45–46.*

Proof. Clearly, while process p is in round r, its task *skip_round* must loop forever in lines 45–46 or in line 48 (otherwise p starts a round higher than r). By the Strong Accuracy property of $\diamond\mathcal{S}_u$, p eventually trusts some process forever and so p cannot loop forever in line 48. Therefore p loops forever in lines 45–46. □

Definition 12. *We say that an eventually-up process* stabilizes *at time t if it recovers at time t and does not crash afterwards. By convention, we say that an always-up process stabilizes at time 0.*

Lemma 46. *Suppose a good process p proposes but never decides. If p receives a message of round r after p stabilizes, then eventually p starts some round $r' \geq r$.*

Proof. In order to obtain a contradiction, suppose that p never starts any round $r' \geq r$. Since p proposes, p starts some round (namely, round 1). Since p does not decide, p blocks in some round $r'' < r$. By Lemma 45, while in round r'', the *skip_round* task of p loops forever in lines 45–46. Since p receives a message of round r after p stabilizes, p eventually exits the loop in lines 45–46. This is a contradiction. □

Lemma 47. *Let p and q be two good processes. If (1) p* s-sends *m to q after p stabilizes, (2) m is the last message p* s-sends *to q, and (3) p never decides, then q receives m from p infinitely often.*

Proof. By (1), (2) and (3), p sends m to q infinitely often in task *retransmit* (line 7). By the Fair Loss property of links, q receives messages from p infinitely often. Note that m is the only message that p sends to q infinitely often: this is because (1) in task *retransmit*, p eventually sends no message different from m to q, and (2) outside task *retransmit*, p can only send messages of the form $(*, \text{DECIDE})$ (line 56); however, such messages are never sent since p never decides. Therefore, by the No Creation and Finite Duplication properties of links, q receives from p only finitely many messages different from m. Since q receives messages from p infinitely often, it follows that q receives m from p infinitely often. □

Lemma 48. *If a good process p blocks in a round r, then the coordinator c of this round is also a good process. Moreover, if $p \neq c$ then c receives messages of round r from p infinitely often.*

Proof. Let p by a good process that blocks in round r and let c be the coordinator of round r. We now prove that c is a good process. In order to obtain a contradiction, suppose that c is bad. Since p blocks in round r, by Lemma 45, while in round r'' the *skip_round* task of p loops forever in lines 45–46. By the Completeness and Monotonicity properties of $\diamond\mathcal{S}_u$, eventually either p permanently suspects c or c's epoch number at p is nondecreasing and unbounded. Therefore, p eventually exits the loop in lines 45–46. This is a contradiction. So c is a good process.

Now, assume $p \neq c$. After p stabilizes, it s-sends a message to c for the last time in round r, either in line 34 or in line 41. By Lemma 47, c receives this message from p infinitely often. □

Lemma 49. *Let p and q be good processes. If p decides and p receives non-*DECIDE *messages from q infinitely often, then eventually q decides.*

Proof. After p decides, every time p receives a non-DECIDE message from q, p sends a DECIDE message to q (line 56). Therefore p sends DECIDE messages to q infinitely often. Moreover, this is the only message that p sends to q infinitely often (since after p decides, it terminates all tasks). This implies that q receives DECIDE messages from p infinitely often. Thus, eventually q decides. □

Lemma 50. *Suppose all good processes propose. If some good process decides then eventually all good processes decide.*

Proof. In order to obtain a contradiction, suppose that every good process proposes and some good process p decides, but there is some good process q that never decides. Let Q be the set of good processes that do not decide. By Lemma 44, for every $q \in Q$, q can start only finitely many rounds. Since q proposes, q blocks in some round r_q. Let $r = \max\{r_q \mid q \in Q\}$, and let $q \in Q$ be a process that blocks in round r.

- *Case 1: q is the coordinator of round r.* Process q never decides, so in round r either q waits forever at line 18 or at line 28 (otherwise q s-sends a DECIDE message to itself in line 30 and then decides in line 53). Before q waits forever, it s-sends a non-DECIDE message to p (line 17 or 27). By Lemma 47, p receives this message infinitely often. By Lemma 49, q eventually decides. This contradicts the fact that $q \in Q$.
- *Case 2: q is not the coordinator of round r.* Let $c \neq q$ be the coordinator of round r. By Lemma 48, c is a good process and c receives messages of round r from q infinitely often. If c decides, then by Lemma 49, q eventually decides too and this contradicts the fact that $q \in Q$. So c never decides. By Lemma 46, eventually c starts a round $r' \geq r$. Since $c \in Q$, by the definition of r, we have that $r' \leq r$. Thus $r' = r$ and so c blocks in round r. By Case 1, c eventually decides — a contradiction. □

Lemma 51. *Suppose there is a majority of good processes. If every good process proposes a value, then eventually some good process decides.*

Proof. In order to obtain a contradiction, suppose that no good process decides. By Lemma 44, each good process p can start only finitely many rounds. Since p proposes, p blocks in some round r_p. Let $r = \max\{r_p \mid p \text{ is good}\}$ and let p be a good process that blocks in round r.

- *Case 1: p is the coordinator of round r.* Process p never decides, so in round r either p waits forever at line 18 or at line 28.
- *Case 1.1: p waits forever at line 18*
  We claim that for every good process q, p eventually receives (r, $estimate_q$, ts_q, ESTIMATE) from q after p stabilizes. Then by the assumption that there is a majority of good processes, p does not wait forever at line 18 — a contradiction.
  To show the claim, note that since p waits forever at line 18 of round r, we have $ts_p \neq r$. Thus, p never updates ts_p to r, and so p never updates $estimate_p$ in round r. By Lemma 40 (1), p never starts Phase NEWESTIMATE. So p never s-sends NEWESTIMATE messages in round r.
- *Case 1.1.1: $q = p$.* Since $ts_p \neq r$, in round r, after p stabilizes and forks task *participant*, p s-sends (r, $estimate_p$, ts_p, ESTIMATE) to itself (line 34). Thus p receives this message after it stabilizes.
- *Case 1.1.2: $q \neq p$.* Before p waits forever at line 18, it s-sends (r, NEWROUND) to q (line 17) after p stabilizes, and this is the last message p s-sends to q. By Lemma 47, q eventually receives (r, NEWROUND) after q stabilizes. By Lemma 46, q eventually starts a round $r' \geq r$. By the definition of r, we have that $r' \leq r$. Thus $r' = r$ and so q starts round r. In round r, we have that $ts_q \neq r$ (otherwise, q sets ts_p to r in line 39, which implies that q received a NEWESTIMATE message from p — contradicting the fact that p never s-sends NEWESTIMATE messages). Then q s-sends message (r, $estimate_q$, ts_q, ESTIMATE) to p (line 34). Process q waits forever in line 35 since p never s-sends a NEWESTIMATE message to q. Therefore (r, $estimate_q$, ts_q, ESTIMATE) is the last message q s-sends to p. By Lemma 47, p eventually receives (r, $estimate_q$, ts_q, ESTIMATE) from q after p stabilizes.
  This concludes the proof of the claim.
- *Case 1.2: p waits forever at line 28*
  We claim that for every good process q, p eventually receives (r, ACK) from q after p stabilizes. Then by the assumption that there is a majority of good processes, p does not wait forever at line 28 — a contradiction.
  We now show the claim.
- *Case 1.2.1: $q = p$.* Before p waits forever at line 28, it s-sends a NEWESTIMATE message to itself (and it does so after p stabilizes). Thus p receives this message from itself. So in task *participant*, p finishes Phase ESTIMATE and s-sends (r, ACK) to itself. Therefore p receives this message from itself after it stabilizes.
- *Case 1.2.2: $q \neq p$.* Before p waits forever at line 28, it s-sends (r, $estimate_p$, NEWESTIMATE) to q and this is the last message p s-sends to q. By Lemma 47, q eventually receives this message from p after q stabilizes. By Lemma 46, q eventually starts a round $r' \geq r$. By the definition of r, we have $r' \leq r$. Thus $r' = r$ and so q blocks in round r. In round r, after q stabilizes and forks task *participant*, q finishes Phase ESTIMATE (since q receives a NEWESTIMATE message from p) and s-sends message (r, ACK) to p in Phase ACK. This is the last message q s-sends to p, since q blocks in round r. By Lemma 47, p eventually receives (r, ACK) from q after p stabilizes.
  This shows the claim.
- *Case 2: p is not the coordinator of round r.*
  Let $c \neq p$ be the coordinator of round r. By Lemma 48, c is a good process and c receives messages of round r from p infinitely often. By Lemma 46, c eventually starts a round $r' \geq r$. By the definition of r, we have that $r' \leq r$. Thus $r' = r$ and so c blocks in round r. In Case 1, we showed that the coordinator of round r does not block in round r — a contradiction. □

Corollary 4 (Termination). *Suppose there is a majority of good processes. If all good processes propose a value, then they all eventually decide.*

Proof. From Lemmata 50 and 51. □

Proof of Theorem 5. Immediate from Lemmata 38 and 43, and Corollary 4. □

Acknowledgements. We would like to thank Rachid Guerraoui, Michel Raynal and André Schiper for introducing us to the problem of consensus in the crash-recovery model, and for explaining their own work on this problem. We are also grateful to Borislav Deianov and the anonymous referees for their helpful comments and suggestions on how to improve the presentation of the results.

References

1. Aguilera MK, Chen W, Toueg S: Heartbeat: a timeout-free failure detector for quiescent reliable communication. In Proceedings of the 11th International Workshop on Distributed Algorithms, Lecture Notes on Computer Science. Springer-Verlag, September 1997. A full version is also available as Technical Report 97-1631, Computer Science Department, Cornell University, Ithaca, New York, May 1997
2. Chandra TD, Hadzilacos V, Toueg S: The weakest failure detector for solving consensus. Journal of the ACM, 43(4):685–722 (1996)
3. Chandra TD, Toueg S: Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 43(2):225–267 (1996)
4. Dolev D, Friedman R, Keidar I, Malkhi D: Failure detectors in omission failure environments. Technical Report 96-1608, Department of Computer Science, Cornell University, Ithaca, New York, September 1996
5. Dwork C, Lynch NA, Stockmeyer L: Consensus in the presence of partial synchrony. Journal of the ACM, 35(2):288–323 (1988)
6. Guerraoui R, Oliveira R, Schiper A: Stubborn communication channels. Technical report, Département d'Informatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland, December 1996
7. Hurfin M, Mostefaoui A, Raynal M: Consensus in asynchronous systems where processes can crash and recover. In Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems, pages 280–286, October 1998
8. Lynch NA: Distributed Algorithms. Morgan Kaufmann Publishers, Inc., 1996
9. Neiger G, Toueg S: Automatically increasing the fault-tolerance of distributed algorithms. Journal of Algorithms, 11(3):374–419 (1990)

10. Oliveira R, Guerraoui R, Schiper A: Consensus in the crash-recover model. Technical Report 97-239, Département d'Informatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland, August 1997

Marcos Kawazoe Aguilera is currently a Ph.D. candidate in the Department Computer Science at Cornell University. He received his B.Eng. in Computer Science from the Universidade Estadual de Campinas in Brazil. His research interests include distributed algorithms and fault-tolerant computing.

Wei Chen received his B.Eng. and M.Eng. degrees from the Department of Computer Science and Technology, Tsinghua University, Beijing, China. He received his M.S. and Ph.D. degrees from the Department of Computer Science, Cornell University in 1998 and 2000, respectively. He is now a senior member of the technical staff at Oracle Corporation. His research interests include distributed systems and fault-tolerant computing.

Sam Toueg received his B.Sc. from the Technion Israel Institute of Technology, and his Ph.D. from the Computer Science Department at Princeton University. He is a professor in the Department Computer Science of Cornell University, which he joined in 1981. He also serves as the Chair of the Departement d'Informatique, at the Ecole Polytechnique in France. His research area is distributed computing.