Failure Detection and Consensus in the Crash-Recovery Model*

Marcos Kawazoe Aguilera Wei Chen Sam Toueg

Cornell University, Computer Science Department, Ithaca NY 14853-7501, USA
aguilera,weichen,sam@cs.cornell.edu

Abstract. 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.

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 [4, 3]. 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 [5, 9, 7] (a brief comparison with these works is in Section 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"

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

S. Kutten (Ed.): DISC'98, LNCS 1499, pp. 231-245, 1998.

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 to be eventually suspected forever [9, 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 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 Section 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.

[1] In fact, this property is assumed in [9, 7].

[2] In [5], 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.

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 [2] 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 [4, 5, 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 [2], we show how to transform any $\diamond\mathcal{S}_e$ to $\diamond\mathcal{S}_u$ 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? Suppose that during each execution of consensus, at least n_a processes are guaranteed to remain up. Clearly, if $n_a < 1$ then consensus cannot be solved: 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 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 [4] can be used).

Is it possible to solve consensus without stable storage if $1 \leq n_a \leq n/2$? To answer this question, assume that in every execution of consensus at most n_b processes are bad. 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 Section 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 $\diamond P$ [4]. There is no contradiction, however, since [4] assumes that all process crashes are

[4] 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.

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 requirement is weaker than $n_a > n_b$).[5]

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.

1.3 Related Work

The problem of solving consensus with failure detectors in systems where processes may recover from crashes was first addressed in [5] (with crash-recovery as a form of omission failures) and more recently studied in [9, 7].

In [5, 7, 9], the question of whether stable storage is always necessary is not addressed, and all the algorithms use stable storage: in [5, 9], 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 [9, 7] use failure detectors that require that unstable processes be eventually suspected forever. The algorithm in [5] 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, and message delays are bounded. 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.

[5] If the good processes are not a majority, a simple partitioning argument as the one in [4] shows that consensus cannot be solved even with $\diamond\mathcal{P}$.

1.5 Roadmap

The paper is organized as follows. Our model is given in Section 2. In Section 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 Section 4. In Section 5, we determine under what conditions consensus requires stable storage. We then give two matching consensus algorithms: one does not require stable storage (Section 6), and the other uses stable storage (Section 7). In Section 8, we briefly consider the performance of these algorithms.

Due to space limitations, all proofs are ommitted here (they are given in [2]).

2 Model

We consider asynchronous message-passing distributed systems. 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.[6] 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.

[6] 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.

2.4 Link Properties

We consider links that do not create messages, or duplicate messages infinitely often. More precisely, we assume that 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.

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, Hurfin *et al.* [7] and Oliveira *et al.* [9] 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 process 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, 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 the following theorem, 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 $\Pi \setminus G$.*

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 [4, 5, 7, 9]. 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 first define $\diamond\mathcal{S}_e$ to be the class of failure detectors $\mathcal{D}$ that satisfy the following properties (the formal definitions of these properties are given in [2]):

- *Monotonicity*: At every good process, eventually the epoch numbers are nondecreasing[7].
- *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.
- *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.

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.

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 [2] 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).

[7] 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.

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* [8], 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 [9, 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 Section 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 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 $\diamond\mathcal{P}$.*

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 [2]. In this paper, 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 Section 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 [2] 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. 1, is based on the rotating coordinator paradigm [4] 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).[8] 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 [7, 9] the burden is entirely on the 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 [5–7]). 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

[8] 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 process p:
1  Initialization:
2    R_p ← ∅; decisionvalue_p ← ⊥; for all q ∈ Π \ {p} do xmitmsg[q] ← ⊥
3  To s-send m to q:
4    if q ≠ p then xmitmsg[q] ← m; send m to q else simulate receive m from p
5  Task retransmit:
6    repeat forever
7      for all q ∈ Π \ {p} do if xmitmsg[q] ≠ ⊥ then send xmitmsg[q] to q
8  upon receive m from q do
9    if m = RECOVERED then R_p ← R_p ∪ {q}
10   if m = (decisionvalue, DECIDE) and decisionvalue_p = ⊥ then
11     decisionvalue_p ← decisionvalue; decide(decisionvalue_p)
12     terminate task {skip_round, 4phases, participant, coordinator, retransmit}
13   if m ≠ (−, DECIDE) and decisionvalue_p ≠ ⊥ then send (decisionvalue_p, DECIDE) to q
14 upon propose(v_p):                          {p proposes v_p via an external input containing v_p}
15   (r_p, estimate_p, ts_p) ← (1, v_p, 0); fork task {4phases, retransmit}
16 Task 4phases:
17   c_p ← (r_p mod n) + 1; fork task {skip_round, participant}
18   if p = c_p then fork task coordinator
19 Task coordinator:
20   {Stage 1: Phase NEWROUND}
21     c_seq_p ← 0
22     repeat
23       PrevR_p ← R_p; c_seq_p ← c_seq_p + 1
24       s-send (r_p, c_seq_p, NEWROUND) to all
25       wait until [ received (r_p, c_seq_p, estimate_q,
26         ts_q, ESTIMATE) from
27         max(n_b + 1, n − n_b − |R_p|) processes ]
28     until R_p = PrevR_p
29     t ← largest ts_q such that p received
30       (r_p, c_seq_p, estimate_q, ts_q, ESTIMATE)
31     estimate_p ← select one estimate_q such that
32       p received (r_p, c_seq_p, estimate_q, t, ESTIMATE)
33     ts_p ← r_p
34   {Stage 2: Phase NEWESTIMATE}
35     c_seq_p ← 0
36     repeat
37       PrevR_p ← R_p; c_seq_p ← c_seq_p + 1
38       s-send (r_p, c_seq_p, estimate_p,
39         NEWESTIMATE) to all
40       wait until [ received (r_p, c_seq_p, ACK) from
41         max(n_b + 1, n − n_b − |R_p|) processes ]
42     until R_p = PrevR_p
43     s-send (estimate_p, DECIDE) to all
44 Task participant:
45   {Stage 1: Phase ESTIMATE}
46     s-send (r_p, WAKEUP) to c_p
47     max_seq_p ← 0
48     repeat
49       if received (r_p, seq, NEWROUND) from c_p
50         for some seq > max_seq_p then
51         s-send (r_p, seq, estimate_p, ts_p,
52           ESTIMATE) to c_p
53         max_seq_p ← seq
54     until [ received (r_p, seq, estimate_{c_p},
55       NEWESTIMATE) from c_p for some seq ]
56     if p ≠ c_p then
57       (estimate_p, ts_p) ← (estimate_{c_p}, r_p)
58   {Stage 2: Phase ACK}
59     max_seq_p ← 0
60     repeat forever
61       if received (r_p, seq, estimate_{c_p},
62         NEWESTIMATE) from c_p for some
63         seq > max_seq_p then
64         s-send (r_p, seq, ACK) to c_p
65         max_seq_p ← seq
66 Task skip_round:
67   d ← D_p                                                  {query ◇S'_e}
68   if c_p ∈ d.trustlist \ R_p then
69     repeat d' ← D_p                                        {query ◇S'_e}
70     until [ c_p ∉ d'.trustlist \ R_p or d.epoch[c_p] < d'.epoch[c_p]
71       or received some message (r, ...) such that r > r_p) ]
72   terminate task {4phases, participant, coordinator}       {abort current round}
73   repeat d ← D_p until d.trustlist \ R_p ≠ ∅                {query ◇S'_e}
74   r_p ← the smallest r > r_p such that [(r mod n) + 1] ∈ d.trustlist \ R_p and
75     r ≥ max{r' | p received (r', ...)}
76   fork task 4phases                                        {go to a higher round}
77 upon recovery:
78   decisionvalue_p ← ⊥; for all q ∈ Π \ {p} do xmitmsg[q] ← ⊥; fork task retransmit
79   s-send RECOVERED to all
```

Fig. 1. Solving Consensus without Stable Storage using $\diamond\mathcal{S}'_e$

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 ($R_c \neq PrevR_c$). If so, c restarts the first stage from scratch.[9] 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 ($R_c \neq PrevR_c$). 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.

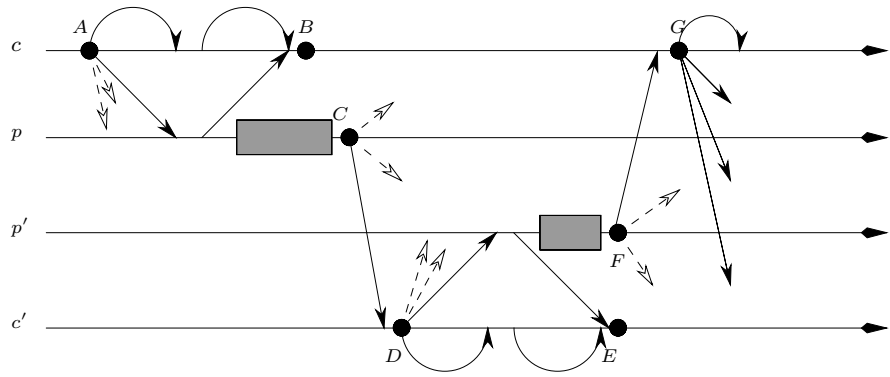
A round r can be interrupted by task *skip_round* (which runs in parallel with tasks *coordinator* and *participant*): a process p aborts its execution of round r if (1) it suspects the coordinator c of round r, or (2) it trusts c but detects an increase in the epoch number of c, or (3) it detects a 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 ($R_c \neq PrevR_c$).

The importance of mechanism (2) is illustrated in Fig. 2: 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

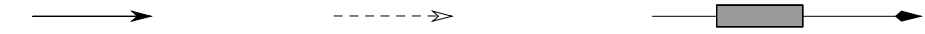
[9] An obvious optimization is for c to check *during the collection of estimates* whether $R_c \neq PrevR_c$. If so it can restart the first stage right away.



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

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 $PrevR_c = \emptyset \neq \{p'\} = R_c$.[10]

Theorem 4. *The algorithm of Fig. 1 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.*

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.

The basic structure of the algorithm (given in Fig. 3) is as in [4, 5] and consists of rounds of 4 phases each (task *4phases*). In each round r, initially the coordinator c

[10] 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.

For every process p:

Initialization:
  for all $q \in \Pi \setminus \{p\}$ **do** $xmitmsg[q] \leftarrow \perp$

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** $xmitmsg[q] \neq \perp$ **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 $r \geq \max\{r' |\ p$ 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** $xmitmsg[q] \leftarrow \perp$
  if propose(v_p) has occurred **and** {check stable storage about propose}
    decide$(-)$ has not occurred **then** {check stable storage about decide}
    retrieve $\{r_p, estimate_p, ts_p\}$
    if $r_p = \perp$ **then** $r_p \leftarrow 1$; **if** $estimate_p = \perp$ **then** $(estimate_p, ts_p) \leftarrow (v_p, 0)$
    fork task {*4phases, retransmit*}

Fig. 3. Solving Consensus with Stable Storage using $\Diamond\mathcal{S}_u$

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 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. 3 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.*

8 Performance of the Consensus Algorithms

8.1 Time and Message Complexity in Nice Runs

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:[11] 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 first algorithm, there are $n-1$ messages for each of the types ESTIMATE, NEWESTIMATE, ACK, and DECIDE; in the second algorithm, there are $n-1$ messages for each of WAKEUP, NEWESTIMATE, ACK, and DECIDE. In contrast, in nice executions the consensus algorithms of [7, 9] reach decision within 2δ and with $O(n^2)$ messages.

[11] This is with the round 1 optimization in which the coordinator chooses its own estimate and sends it without waiting for estimates from other processes.

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).

Remark The full version of this paper [2] contains the following additional material: a consensus algorithm that does not require stable storage and uses $\diamond\mathcal{S}_e$ (rather than $\diamond\mathcal{S}'_e$), an implementation of $\diamond\mathcal{S}_e$ and $\diamond\mathcal{S}'_e$ in some models of partial synchrony, an algorithm that transforms $\diamond\mathcal{S}_e$ into $\diamond\mathcal{S}_u$, a discussion on how to do repeated consensus, the formal definition of the failure detector properties, and all the proofs.

Acknowlegments 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 would also like to thank Borislav Deianov for his helpful comments on an earlier draft.

References

1. M. K. Aguilera, W. Chen, and S. Toueg. 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, Sept. 1997. A full version is also available as Technical Report 97-1631, Computer Science Department, Cornell University, Ithaca, New York, May 1997.
2. M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. Technical Report 98-1676, Department of Computer Science, Cornell University, April 1998.
3. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
4. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
5. D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. Technical Report 96-1608, Department of Computer Science, Cornell University, Ithaca, New York, Sept. 1996.
6. R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels. Technical report, Département d'Informatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland, Dec. 1996.
7. M. Hurfin, A. Mostefaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. Technical Report 1144, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, Nov. 1997.
8. G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.
9. R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report 97-239, Département d'Informatique, Ecole Polytechnique Fédérale, Lausanne, Switzerland, Aug. 1997.