Fast Indulgent Consensus with Zero Degradation*

Partha Dutta Rachid Guerraoui
Distributed Programming Laboratory
Swiss Federal Institute of Technology in Lausanne

Abstract

This paper describes a new consensus algorithm for the asynchronous message passing system model augmented with an unreliable failure detector abstraction: channels are reliable, processes can fail by crashing, and the detection of crashes are not reliable. Our algorithm (a) matches all known consensus lower bounds on (1) *failure detection, i.e.,* Ω, (2) *resilience*, i.e., a majority of correct processes, and (3) *latency*, i.e., two communication steps for a global decision in *nice* runs (when no process crashes and the failure detection is reliable), and (b) has the following *zero degradation* flavor: in every *stable* run of the algorithm (when all failures are initial crashes, and failure detection is reliable), two communication steps are sufficient to reach a global decision.

The zero degradation flavor is particularly important when consensus is used in a repeated form: failures in one consensus instance do not impact performance of future consensus instances.

We describe our algorithm informally as well as in a detailed way, and then we give two additional variants of our algorithm, both preserving the zero degradation flavor: (1) a variant that reaches a global decision in one communication step in stable runs if all correct processes propose the same privileged value, and (2) a variant that uses a $\diamond\mathcal{S}$ failure detector (instead of Ω). Roughly speaking, these variants convey the very fact that our technique to obtain zero degradation can be applied in various contexts.

1 Introduction

1.1 The Motivation

In practice, most runs of a distributed system are *nice*: failures are rare and failure detectors do not usually suspect correct processes to have crashed. Hence, it is important to optimize the performance of distributed algorithms in nice runs. We are interested here in also minimizing the performance degradation of the algorithms in the presence of failures, especially initial failures. To see why this is also important, consider a long-running application using a series of instances of a given distributed algorithm (e.g., an atomic broadcast service using a series of consensus instances [Lam89, CT96]). Even if failures are rare, they might occur and one expects a failure to impact the performance of the instance of the algorithm during which the failure occurs. However, it is desirable to minimize the impact of that failure on all subsequent instances of the algorithm. If this impact is nil, then we say that the algorithm has a "zero degradation" flavor.[1]

*This work is supported by the Swiss National Science Foundation (NSF).

[1]When considering this flavor, we focus on the time complexity of an algorithm, and more precisely on its latency, i.e., the number of communication steps needed for all correct processes to decide (a *global decision*). Aspects like message or memory complexity are outside the scope of this paper.

Our motivation here is to devise a consensus algorithm that, on the one hand, matches all known consensus lower bounds on (1) *failure detection, i.e.,* Ω [CHT96], (2) *resilience*, i.e., a majority of correct processes [CT96], and (3) *latency*, i.e., two communication steps for a global decision in *nice* runs [KR01], and on the other hand, provides the zero degradation flavor.

1.2 The Background

The consensus problem consists for a set of processes to decide a common value, among one of the values proposed by the processes. Each process proposes a value v through procedure propose(v). If propose($*$) returns v' at a process, the process is said to have decided v'. Consensus is solved if the following three conditions are ensured: (1) (*validity*) if a process decides v then some process has proposed v, (2) (*uniform agreement*) no two processes decide differently,[2] and (3) (*termination*) every correct process eventually decides.

We consider consensus in a message-passing distributed system model consisting of a set of n processes: $\Pi = \{p_1, p_2, ..., p_n\}$. Processes can fail by crashing and never recover from a crash.[3] A *correct* process is a process that never crashes and executes the deterministic algorithm assigned to it. A process that crashes is said to be *faulty*. Any pair of processes can communicate through *send* and *receive* primitives, which emulate a *reliable* communication channel in the following sense [HT93]: (1) any message sent to a correct process is eventually received, (2) no message is received more than once, and (3) the channel does not create or alter messages. To solve consensus in this model [FLP85, DDS87], one needs to consider additional assumptions such as a majority of the processes is correct and the system is "eventually synchronous". The latter assumption can be captured in a modular manner through the abstraction of a failure detector oracle that provides the processes with some (possibly unreliable) information about which process has crashed and which process has not [CT96].

In this paper, we consider consensus algorithms based on two interesting failure detectors: Ω and $\diamond\mathcal{S}$. The failure detector Ω outputs at each process, a (leader) process denoted $\Omega.trusted$ such that, eventually, at all correct processes the output is the same correct process. The failure detector $\diamond\mathcal{S}$ outputs, at each process, a list of suspected processes denoted $\diamond\mathcal{S}.suspected$ such that: (1) (*strong completeness*) eventually, every crashed process is permanently suspected by every correct process, and (2) (*eventual weak accuracy*) there is a time after which some correct process is never suspected by any correct process.

In [CT96], Chandra and Toueg presented a consensus algorithm (which we denote by CT) assuming a majority of correct processes and the abstraction of $\diamond\mathcal{S}$. Independently, Lamport presented in [Lam89] the Paxos consensus algorithm (which we denote by PC) assuming a majority of correct processes and the abstraction of Ω.[4] Both failure detectors were shown to be equivalent in a precise sense and represent the amount of knowledge needed to solve consensus [CHT96], i.e., a *failure detection* lower bound. An inherent characteristic of Ω and $\diamond\mathcal{S}$ is the *indulgence* of the actual consensus algorithms using them [Gue00]. Roughly speaking, the algorithm is *indulgent* towards its failure detector: even if this failure detector turns out to be completely unreliable and does never provide any useful knowledge about failures (i.e., the sys-

[2] Note that we consider the *uniform* consensus problem. In the system model we consider, uniform consensus and consensus are similar [Gue00].

[3] Applying our ideas to the crash-recovery model of [ACT00] is certainly feasible but might distract from the main ideas we are addressing here: achieving zero degradation while matching consensus lower bounds.

[4] In fact, PC was devised for a system model where channels might lose messages and processes can crash and recover. For the sake of presentation simplicity, we consider a variant of the algorithm in the simpler system model of [CT96]. In this model, the eventual synchrony assumption of Paxos Consensus can be captured through the failure detector Ω.

tem does never provide any synchrony guarantee), the safety properties of consensus (validity and agreement) are preserved. It is shown in [CT96, Gue00] that a majority of correct processes is a lower bound for this form of indulgence, i.e., a *resilience* lower bound.

1.3 The Question

Precisely because of the indulgence of algorithms using Ω and $\Diamond\mathcal{S}$, one cannot bound the number of communication steps needed to reach a global decision (*latency*). Fortunately, it is possible to bound this latency when the failure detector does not make mistakes, in particular in *stable* runs. Intuitively, we say that a run is stable if failures are initial (i.e., all failures occurred before the run started) and the failure detector output does not change during the run. More precisely, we say that a run of an Ω-based algorithm is *stable* iff all failures in the run are initial failures and the failure detector outputs same correct process, at all processes, from the very beginning. Similarly, we say that a run of a $\Diamond\mathcal{S}$-based consensus algorithm is stable iff all failures in the run are initial failures, and at all processes, $\Diamond\mathcal{S}.suspected$ is always identical to the set of initially crashed processes. A *nice* run is simply a stable run with no failures.

In nice runs of CT, four communication steps are needed before a consensus decision is reached by all correct processes (global decision). One can easily obtain an optimization of CT that alleviates the need for the first step in a nice run. In every *stable* run of CT, the same number of communication steps (four) are still needed for a global decision. Similarly, in nice runs of PC, five communication steps are needed for a global decision, and a simple optimization of PC alleviates the need for the first two steps in a nice run. In every stable run, still the same number of communication steps (five) are needed. In other words, though the latency in stable runs is relatively high, it does not depend on the identity or the number of the initially failed processes.

Several authors suggested variants of CT where two communication steps are sufficient for a global decision in nice runs [Sch97, HR99]: a latency lower bound for these runs [KR01]. Unlike CT, these algorithms degrade in the presence of (initial) failures; the degradation being more or less graceful depending on algorithm specifics [HR99]. More recently, [MR01] presented two Ω-based consensus algorithms that do not degrade in stable runs. In the first algorithm of [MR01], three communication steps are required for a global decision in stable runs, thus clearly not optimum in latency. The second algorithm enforces global decision in two communication steps in stable runs but assumes two-third of the processes to be correct: thus clearly not optimum in resilience.

We say that a consensus algorithm is *zero degrading* iff the same number of communication steps are required for achieving a global decision in every stable run (irrespective of the identity or the number of the initially crashed processes). To our knowledge, previous indulgent consensus algorithms that have an optimal latency in nice runs, either are not zero degrading, or are not optimal in terms of resilience. It is legitimate to ask whether we can have the cake and eat it too. This paper shows that the answer is *yes*: we can indeed match (1) the lower bounds on resilience, failure detection, and latency, and (2) yet provide the zero degradation flavor.

1.4 The Contribution

We describe a consensus algorithm based on the assumptions of failure detector Ω and a majority of correct processes. In every stable run (whether nice or not), two communication steps are sufficient to reach a global decision. Our algorithm is decentralized: processes exchange consensus *decisions* and *estimates* of the decisions directly, just like in [Sch97, HR99, MR01]. What makes

```
at process p_i
propose(v_i)
    start Task 1; start Task 2

    Task 1
        r_i ← 0; estimate_i ← v_i; newEstimate_i ← ⊥; leader_i ← ⊥
        while(true)
            leader_i ← Ω.trusted; newEstimate_i ← ⊥
            send(ESTIMATE, r_i, estimate_i, leader_i) to Π
            wait until ((received(ESTIMATE, r_i, *, *) from leader_i and ⌈(n+1)/2⌉ − 1 other processes) or
            (leader_i ≠ Ω.trusted))
            if ((received(ESTIMATE, r_i, *, leader_i) from leader_i) and
            (received(ESTIMATE, r_i, *, leader_i) from ⌈(n+1)/2⌉ − 1 other processes)) then
                newEstimate_i ← (estimate received from leader_i)
            send(NEWESTIMATE, r_i, newEstimate_i) to Π
            wait until received(NEWESTIMATE, r_i, *) from ⌈(n+1)/2⌉ processes
            if (received(NEWESTIMATE, r_i, newEstimate) s.t. newEstimate ≠ ⊥ from ⌈(n+1)/2⌉ processes) then
                estimate_i ← (newEstimate of any received NEWESTIMATE message)
                send(DECIDE, estimate_i) to Π\p_i; return(estimate_i)                    {Decision}
            else if (received any (NEWESTIMATE, r_i, newEstimate') s.t. newEstimate' ≠ ⊥) then
                estimate_i ← newEstimate'
            r_i ← r_i + 1

    Task 2
        upon receiving (DECIDE, x)
            send(DECIDE, x) to Π\p_i
            return(x)                                                                    {Decision}
```

Figure 1: The consensus algorithm DG_Ω

our algorithm particularly effective is the very fact that processes also exchange their perception about the current leader. Intuitively, they can expedite the decision when they realize that they have the same leader, e.g., in a stable run.

Section 2 gives an overview and then a detailed description of the algorithm with an informal argument for its correctness. For space limitation, the detailed correctness proofs of our algorithms are given in the optional appendix. We then briefly describe in Section 3 and in Section 4 two additional variants of our algorithm: (1) an Ω-based consensus algorithm such that, given a privileged value PV, one communication step is sufficient to reach consensus in every stable run where all processes propose PV, and (2) a zero degrading $\diamond\mathcal{S}$-based consensus algorithm. Roughly speaking, these variants convey the very fact that our technique to obtain zero degradation can be applied in various indulgent consensus context.

As a side effect of our work, we introduce in Section 5 a performance metric for consensus algorithms which captures the best-case latency of an algorithm (i.e., latency in a nice run) as well as reveals the performance degradation (if any) of the algorithm in the presence of failures. We use this metric to compare our algorithms with previous indulgent consensus algorithms. We point out, finally, the performance gain obtained by our algorithm when consensus is used in a repeated form, with respect to traditional consensus algorithms.

2 The Algorithm

We denote our algorithm by DG_Ω and present it in Figure 1. We give here a description of the algorithm along with an informal argument of its correctness. Detailed correctness proofs are given in the appendix.

2.1 Overview

Our DG_Ω algorithm is round based: every process p_i moves incrementally from one round to the other. Every round consists of two phases; each phase involves exchanging a set of messages. Unless p_i decides (returns from propose($*$)), p_i moves to the next higher round after completing the two phases.

At each round r, every process queries its Ω failure detector module about the current leader. We say that a process p_i is a *majority-leader* at a round r iff p_i is the current leader at a majority of processes at that round. For a given round r, there can obviously be at most one majority-leader. If the failure detector makes mistakes, then it is possible that there is no majority-leader at a round. In the first phase of a round, processes exchange current leader values; i.e., they exchange the perception about who is the leader. If a process perceives that a majority-leader exists for round r, it adopts the *estimate* of that leader, say value x, as its intermediate *estimate* value, $newEstimate$; otherwise $newEstimate$ remains $\perp$. Since there is at most one majority-leader in a round, $newEstimate$ at every process is either x or $\perp$.

Due to the unreliability of the failure detector and process failures, some processes may perceive that a majority-leader exists at round r, whereas some processes may not. Therefore, in the second phase of a round, processes exchange $newEstimate$ values. On receiving $newEstimate$ values from a majority of processes, if a process receives $newEstimate = x$ from all processes in that majority, then the process decides x. If a process receives both $newEstimate = x$ and $newEstimate = \perp$, then it sets its *estimate* to x. If all received values are $\perp$, then a process does not update its *estimate*. If any process decides x (by receiving a majority of $newEstimate = x$), then clearly every process receives at least one message with $newEstimate = x$ and hence, updates *estimate* to x. We now give a detailed description of the algorithm.

2.2 Description

The algorithm consists of two parallel tasks: Task 1 and Task 2. When a process proposes a value, it starts both tasks. The execution terminates when the *propose* function returns a value (from Task 1 or from Task 2). We now describe the two tasks.

Task 1: This task proceeds in asynchronous rounds with processes incrementally moving from one round to the other. Each round has two phases: (*phase 1*) exchanging ESTIMATE messages, and (*phase 2*) exchanging NEWESTIMATE messages. Consider any process p_i: p_i maintains (1) the current round number r_i, initialized to 0, (2) an estimate of the possible decision value $estimate_i$, which is initialized to the input value of p_i, and (3) an intermediate $newEstimate_i$ value (a possible new value for $estimate_i$), initialized to $\perp$ at the beginning of each round. Further, at the beginning of each round, p_i queries Ω about the current leader and stores the identity of that leader in $leader_i$. Once $leader_i$ is set at the beginning of a round, it does not change inside the round (even if $\Omega.trusted$ changes).

At the beginning of a round, p_i sends ESTIMATE messages to all processes containing $estimate_i$ and $leader_i$. Process p_i waits till it receives ESTIMATE messages from $leader_i$ and $\lceil \frac{n+1}{2} \rceil - 1$ other processes. It simultaneously keeps on querying Ω. The value of $newEstimate_i$ depends on the output of Ω and the ESTIMATE messages received:

1. If $leader_i \neq \Omega.trusted$ before ESTIMATE message from $leader_i$ is received by p_i, or any of the $\lceil \frac{n+1}{2} \rceil$ PROPOSAL messages received by p_i has $leader \neq leader_i$, then $newEstimate_i$ remains $\perp$.

2. If p_i received ESTIMATE messages from $leader_i$ and $\lceil\frac{n+1}{2}\rceil - 1$ other processes, and every received message has $leader = leader_i$, then $newEstimate_i$ is set to the *estimate* received from $leader_i$.

In the second phase of the round, p_i sends a NEWESTIMATE message, containing $newEstimate_i$, to all processes. Process p_i waits till it receives NEWESTIMATE messages from $\lceil\frac{n+1}{2}\rceil$ processes and then takes one of the following three steps depending on the received messages:

1. If every NEWESTIMATE message received by p_i has $newEstimate \neq \perp$, then p_i adopts any received *newEstimate* as $estimate_i$. Afterwards, p_i sends a DECIDE message with $estimate_i$ as the decision value to all processes different from p_i, and returns $estimate_i$ (i.e., decides $estimate_i$).

2. If any NEWESTIMATE message received by p_i has $newEstimate \neq \perp$, then p_i adopts that *newEstimate* as $estimate_i$. Afterwards, p_i proceeds to the next round.

3. If every NEWESTIMATE message received by p_i has $newEstimate = \perp$, then p_i proceeds to the next round (without updating $estimate_i$).

Task 2: Upon receiving a DECIDE message with value x, p_i sends a DECIDE message with x as the decision value to all processes different from p_i, and returns x (i.e., decides x).

2.3 Correctness

We now informally argue about the correctness of the algorithm. (The complete proof is given in the appendix.) Validity is straightforward and termination is guaranteed by the presence of an Ω failure detector and a majority of correct processes. The heart of the algorithm deals with agreement.

If some process decides (i.e., returns from propose($*$)) then some process must have sent a decision value at line 15. Consider the smallest round, say r, in which some decision value is sent at line 15. Assume process p_i sends decision value v at line 15 of round r. Notice that if some process sends v as the decision value, then it must have received a NEWESTIMATE message with $newEstimate = v$ from some process, say p_j. Let p_l be the *leader* at p_j at round r. By the algorithm, all $\lceil\frac{n+1}{2}\rceil$ ESTIMATE messages received by p_j must have $leader = p_l$. Therefore, every process which receives a majority of ESTIMATE messages must have received at least one message with $leader = p_l$. So, at processes where $leader \neq p_l$, *newEstimate* remained $\perp$, and at processes where $leader = p_l$, either v was adopted as *newEstimate* or *newEstimate* remained $\perp$ (line 9). Therefore, every process which exchanged ESTIMATE messages has $newEstimate \in \{v, \perp\}$ before it sends a NEWESTIMATE message.

Further, before sending decision value v at line 15, p_i must have received $newEstimate = v$ from $\lceil\frac{n+1}{2}\rceil$ processes. Therefore, every process which completes round r must have received $newEstimate = v$ from at least one process (since completion of round r requires receiving NEWESTIMATE messages from a majority). Since, *newEstimate* values sent at round r are restricted to $\{v, \perp\}$, no NEWESTIMATE message is received with $newEstimate \notin \{v, \perp\}$. Therefore, every process which completes round r adopts v as its *estimate* (line 17). Similarly, we can show that every decision value sent at round r is v.

Clearly, there are no *estimate* values different from v after round r. Thus, no decision value sent at line 15 of a round higher than r can be different from v. Since r is the smallest round in which some decision value is sent at line 15, then *every decision message has the same value, v.*

2.4 Zero degradation

Consider any stable run of DG_Ω, i.e., any run where (1) all faulty processes crash before the consensus run starts, and (2) there is a correct process p_c such that, at every correct process, $\Omega.trusted$ is always p_c. Let v be the value proposed by process p_c. Every correct process sends a (ESTIMATE, 0, $*$, p_c) message to all processes. Correct processes receive (ESTIMATE, 0, $*$, p_c) from $\lceil\frac{n+1}{2}\rceil$ processes, including a (ESTIMATE, 0, v, p_c) message from p_c. Thus, correct processes adopt v as $newEstimate$. Then, every correct process sends (NEWESTIMATE, 0, v) to all processes. On receiving (NEWESTIMATE, 0, v) from $\lceil\frac{n+1}{2}\rceil$ processes, correct processes send (DECIDE, v) and decide v. Thus, in every stable run of A, all decide events occur in two communication steps. Note that, every nice run is a stable run, and hence, in every nice run of A, all decide events occur in two communication step.

3 One-step Consensus with Zero Degradation

In a stable period, every run of our consensus algorithm terminates in two communication steps. Can we do better? The answer is "sometimes, yes" . The Ω-consensus lower bound [KR01] actually means that every Ω-based consensus algorithm has a nice run where at least one correct process needs at least two steps to decide. In fact the lower bound does not preclude the existence of an Ω-based consensus algorithm where, from any starting configuration in a specific non-empty subset of initial configurations, all correct processes need only one step to decide in every nice run.

We briefly describe below a simple variant of our DG_Ω consensus algorithm, denoted DG'_Ω and given in Figure 2. In addition to the assumptions of DG_Ω, we assume for DG'_Ω that all processes have an a priori knowledge of a *privileged value,* PV. Just like for DG_Ω, in every stable run of DG'_Ω, two communication steps are sufficient for all correct processes to decide. Moreover, in all stable runs of DG'_Ω where all correct processes propose PV, one communication step is actually sufficient.

To obtain DG'_Ω, we apply to DG_Ω an idea borrowed from [BGMR01]. Only the first round of DG_Ω ($r_i = 0$) is modified. In this first round, if a process p_i receives (ESTIMATE, 0, PV, $leader_i$) messages from $leader_i$ as well as $\lceil\frac{n+1}{2}\rceil - 1$ other processes, p_i sends PV as the decision value to all and decides PV . Otherwise, p_i waits till it receives $\lceil\frac{n+1}{2}\rceil$ ESTIMATE messages, and if p_i received any ESTIMATE message with $estimate$ = PV then p_i adopts PV as its $estimate$. This idea is conveyed in lines 11-17 of Figure 2 (the main difference with Figure 1). In a stable run, if all correct processes propose PV, then every process receives $estimate$ = PV from its $leader$ and $\lceil\frac{n+1}{2}\rceil - 1$ other processes in round 0. Thus, all correct processes send (DECIDE, PV) and decide in one communication step. In any stable run, even if all processes do not propose PV, processes decides in two communication steps. Thus, DG'_Ω retains the zero degradation flavor of DG_Ω.

Similar to DG_Ω, the heart of DG'_Ω deals with preserving agreement. The algorithm ensures that if any process sends (DECIDE, v) in round 0, then (i) any process which starts round 1 has $estimate = v$, and (ii) any (DECIDE, v') message at round 0 has $v' = v$. This is sufficient to ensure agreement, since in all subsequent rounds, DG'_Ω is identical to DG_Ω. A sketch of the correctness proofs is given in the appendix.

```
at process p_i
01: propose(v_i)
02:     start Task 1; start Task 2

03:     Task 1
04:         r_i ← 0; estimate_i ← v_i; newEstimate_i ← ⊥; leader_i ← ⊥
05:         while(true)
06:             leader_i ← Ω.trusted; newEstimate_i ← ⊥
07:             send(ESTIMATE, r_i, estimate_i, leader_i) to Π
08:             wait until ((received(ESTIMATE, r_i, *, *) from leader_i and ⌈(n+1)/2⌉ − 1 other processes) or
                (leader_i ≠ Ω.trusted))
09:             if ((received(ESTIMATE, r_i, *, leader_i) from leader_i) and
                (received(ESTIMATE, r_i, *, leader_i) from ⌈(n+1)/2⌉ − 1 other processes)) then
10:                 newEstimate_i ← (estimate received from leader_i)
11*:             if (r_i = 0) then
12*:                 if (received(ESTIMATE, 0, PV, leader_i) from (leader_i and ⌈(n+1)/2⌉ − 1 other processes)) then
13*:                     send(DECIDE, PV) to Π\p_i; return(PV)                                        {Decision}
14*:                 if (number of (ESTIMATE, 0, *, *) received < ⌈(n+1)/2⌉) then
15*:                     wait until (received(ESTIMATE, 0, *, *) from ⌈(n+1)/2⌉ processes)
                                                            {Including the messages received at line 8}
16*:                 if (received a (ESTIMATE, 0, PV, *)) then
17*:                     estimate_i ← PV
18:             send(NEWESTIMATE, r_i, newEstimate_i) to Π
19:             wait until received(NEWESTIMATE, r_i, *) from ⌈(n+1)/2⌉ processes
20:             if (received(NEWESTIMATE, r_i, newEstimate) s.t. newEstimate ≠ ⊥ from ⌈(n+1)/2⌉ processes) then
21:                 estimate_i ← (newEstimate of any received NEWESTIMATE message)
22:                 send(DECIDE, estimate_i) to Π\p_i; return(estimate_i)                        {Decision}
23:             else if (received any (NEWESTIMATE, r_i, newEstimate') s.t. newEstimate' ≠ ⊥) then
24:                 estimate_i ← newEstimate'
25:             r_i ← r_i + 1

26:     Task 2
27:         upon receiving (DECIDE, x)
28:             send(DECIDE, x) to Π\p_i
29:             return(x)                                                                    {Decision}
```

Figure 2: The consensus algorithm DG'_{Ω}

at process p_i
01: propose(v_i)
02: **start Task 1**; **start Task 2**

03: **Task 1**
04: $r_i \leftarrow 0$; $estimate_i \leftarrow v_i$; $newEstimate_i \leftarrow \perp$; $leader_i \leftarrow \perp$
05*: $leader_i \leftarrow$ process with the lowest index in $\{\Pi$ - $\Diamond\mathcal{S}.suspected\}$; $newEstimate_i \leftarrow \perp$
06: send(ESTIMATE, r_i, $estimate_i$, $leader_i$) to Π
07*: **wait until** ((received(ESTIMATE, r_i, $*$, $*$) from $leader_i$ and $\lceil \frac{n+1}{2} \rceil - 1$ other processes) **or** ($leader_i \in \Diamond\mathcal{S}.suspected$))
08: **if** ((received(ESTIMATE, r_i, $*$, $leader_i$) from $leader_i$) **and** (received(ESTIMATE, r_i, $*$, $leader_i$) from $\lceil \frac{n+1}{2} \rceil - 1$ other processes)) **then**
09: $newEstimate_i \leftarrow$ ($estimate$ received from $leader_i$)
10: send(NEWESTIMATE, r_i, $newEstimate_i$) to Π
11: **wait until** received(NEWESTIMATE, r_i, $*$) from $\lceil \frac{n+1}{2} \rceil$ processes
12: **if** (received(NEWESTIMATE, r_i, $newEstimate$) *s.t.* $newEstimate \neq \perp$ from $\lceil \frac{n+1}{2} \rceil$ processes) **then**
13: $estimate_i \leftarrow$ ($newEstimate$ of any received NEWESTIMATE message)
14: send(DECIDE, $estimate_i$) to $\Pi \backslash p_i$; return($estimate_i$) {Decision}
15: **else if** (received any (NEWESTIMATE, r_i, $newEstimate'$) *s.t.* $newEstimate' \neq \perp$) **then**
16: $estimate_i \leftarrow newEstimate'$
17*: return(propose$_C$($estimate_i$)) {Decision}

18: **Task 2**
19: **upon** receiving (DECIDE, x)
20: send(DECIDE, x) to $\Pi \backslash p_i$
21: return(x) {Decision}

Figure 3: The consensus algorithm $DG_{\Diamond\mathcal{S}}$

4 A $\Diamond\mathcal{S}$-based Zero Degrading Algorithm

We now discuss how DG_Ω can be transformed to a $\Diamond\mathcal{S}$-based algorithm $DG_{\Diamond\mathcal{S}}$ which retains the zero degradation flavor; i.e., in every stable run, $DG_{\Diamond\mathcal{S}}$ achieves a global decision in two communication steps. The algorithm is given in Figure 3. For simplicity of presentation, we assume an independent $\Diamond\mathcal{S}$-based consensus algorithm C accessed through procedure propose$_C(*)$ which returns the decision value (e.g., the $\Diamond\mathcal{S}$-based consensus algorithm of [CT96]). Irrespective of the time complexity of C, $DG_{\Diamond\mathcal{S}}$ achieves a global decision in two communication steps in every stable run.

The first round of $DG_{\Diamond\mathcal{S}}$ follows nearly the same pattern as that of DG_Ω. If a process is unable to decide in the first round then it invokes propose$_C(*)$ to obtain the decision value (line 17). The primary difference between the first round of $DG_{\Diamond\mathcal{S}}$ and that of DG_Ω is in the selection of the current leader. In $DG_{\Diamond\mathcal{S}}$, the current leader at a process is the process with the lowest index in Π - $\Diamond\mathcal{S}.suspected$ (line 5). In any stable run, the set Π - $\Diamond\mathcal{S}.suspected$ is precisely the set of correct processes, and hence, the current leader is the same correct process at all correct processes.[5] Using arguments similar to Section 2.4 for DG_Ω, one can easily show that every stable run of $DG_{\Diamond\mathcal{S}}$ reaches a global decision in two communication steps.

[5] Obviously, $\Diamond\mathcal{S}$ does not guarantee that "the process with the lowest index in Π - $\Diamond\mathcal{S}.suspected$ is the same correct process at all correct processes". The claim is true only for stable runs.

5 Performance

5.1 Time complexity metric

To measure the time complexity of our algorithms and compare it with other consensus algorithms, we introduce a metric denoted $cs_{F,A}$, which captures the number of communication steps of a consensus algorithm A in a given failure pattern F. The metric was informally introduced in [HR99]. We define it more precisely using a variant of Lamport's logical clock:

- *Modified logical clock:* Consider Lamport's logical clock ([Lam78]), as modified in [Sch97]: (1) send and local events at a process do not change the logical clock, and (2) the time-stamp of a receive(m) event at p_i is: maximum{(time-stamp of send(m) at $sender(m)+1$), (time-stamp of the event preceding receive(m) at p_i)}.

We then introduce the following notations:

- cs_R: The number of communication steps of a consensus run R is the largest time-stamp of all *decide* events in that run.
- $cs_{C,F,A}$: The number of communication steps of a consensus algorithm A in a failure pattern F and an initial configuration C,[6] is the smallest cs_R of all runs of A with initial configuration C and failure pattern F.
- $cs_{F,A}$: The number of communication steps of a consensus algorithm A in a failure pattern F, is the largest $cs_{C,F,A}$ of all possible initial configurations of A with failure pattern F.

The $cs_{F,A}$ metric captures the performance of an algorithm in nice runs, as well as the degradation of performance in the presence of failures. By selecting the fastest run among all possible runs with the same initial configuration C and failure pattern F, we eliminate the effect of unreliable failure detection. Further, by choosing the maximum among all $cs_{C,F,A}$ with the same F, the metric does not advantage algorithms that are particularly efficient for specific initial configurations (e.g., our algorithm DG'_Ω).

5.2 Performance

Consider algorithms DG_Ω and $DG_{\diamond\mathcal{S}}$. As we show in Section 2.4, there exists runs of each algorithms with initially crashed processes in which all decide events occur within two communication steps, irrespective of the initial configuration. Thus for any failure pattern F in which all faulty processes crash at $t=0$ (i.e., before the consensus run starts), $cs_{F,DG_\Omega} = cs_{F,DG_{\diamond\mathcal{S}}} = 2$.

In case of DG'_Ω, notice that for every initial configuration C in which no process proposes PV, every run of DG'_Ω which starts from C requires at least two communication steps for global decision. Thus, even though DG'_Ω reaches a global decision in one communication step for some initial configuration, for any failure pattern F, $cs_{F,DG'_\Omega} \geq 2$. (Obviously, for any failure pattern F in which all faulty processes crash at $t=0$, $cs_{F,DG'_\Omega} = 2$.)

5.3 Comparisons

Table 1 compares the performance of DG_Ω and $DG_{\diamond\mathcal{S}}$ with alternative indulgent consensus algorithms that tolerate a minority of failures. We consider a system of at least 7 processes

[6] The initial configuration of a distributed system is defined by the initial state of each process and empty communication channels [FLP85]. Here, we are specifically interested in the list of proposed values.

	F0	*F1*	*F2*	*F3*
$\Diamond\mathcal{S}$-based consensus algorithms				
CT	3	4	4	4
SC	2	4	6	8
HR	2	3	4	5
$\mathrm{DG}_{\Diamond\mathcal{S}}$	2	2	2	2
Ω-based consensus algorithms				
PC	3	5	5	5
DPC	2	4	4	4
MR	3	3	3	3
DG_{Ω}	2	2	2	2

Table 1: $cs_{F,A}$ values

($n \geq 7$), and the following failure patterns: (i) *F0:* all processes are correct; (ii) *F1:* Process p_1 crashes at $t = 0$ and all other processes are correct; (iii) *F2:* Processes p_1 and p_2 crash at $t = 0$ and all other processes are correct; and (iii) *F3*: Processes p_1, p_2 and p_3 crash at $t = 0$ and all other processes are correct.

We consider three $\Diamond\mathcal{S}$-based algorithms, CT: Chandra-Toueg's original $\Diamond\mathcal{S}$ consensus algorithm [CT96], SC: early consensus [Sch97], and HR: fast consensus [HR99], and compare them in Table 1 with with our $\Diamond\mathcal{S}$-based algorithm, $DG_{\Diamond\mathcal{S}}$. The $cs_{F,A}$ values are achieved in any stable run; i.e., when all process crashes are initial and, throughout the run, at every correct process, the suspicion list of the failure detector is identical to the set of initially crashed processes.

Besides our algorithm (DG_{Ω}), we also consider three Ω-based algorithms; PC: Lamport's Paxos Consensus [Lam89], MR: the first algorithm in [MR01] (we do not consider here the second algorithm of [MR01] because it assumes at least two-thirds of the processes are correct, and is hence incomparable with other algorithms), and DPC: a decentralized version of PC [Lam89], pointed out in [KR01]. The $cs_{F,A}$ values are achieved in any stable run, i.e., when all process crashes are initial and, throughout the run, the same correct process remains the leader at every correct process.[7]

The $cs_{F,A}$ values are summarized in Table 1, which clearly conveys the efficiency of DG_{Ω} and $DG_{\Diamond\mathcal{S}}$. In short, apart from achieving the failure-free performance of SC and MR ($F0$ failure pattern, i.e., nice runs), DG_{Ω} and $DG_{\Diamond\mathcal{S}}$ are immune to the presence of crashed processes in stable runs (zero degradation). It is important to notice here that, similar to the algorithms of [Sch97, HR99, MR01], the message complexity of DG_{Ω} and $DG_{\Diamond\mathcal{S}}$ are $O(n^2)$ when the processes are connected by a point-to-point network, and $O(n)$ in a broadcast network. The original CT

[7] A round in the Paxos consensus algorithm (PC) can be divided into three phases: (i) *read phase:* the leader (elected by some leader election service; e.g., Ω) reads whether any *estimate* value might already be locked at a majority of processes, (ii) *write phase:* the leader tries to lock an *estimate* value at majority of processes, and (iii) *decide phase*: the leader disseminates the successfully locked *estimate* as the decision value. The read and the write phase each requires two communication steps (messages sent by the leader to all processes and the processes sending acknowledgments to the leader), and the decide phase requires one communication step. Optimizations can be made along the lines proposed by [KR01]. In a crash-stop model, the read phase of the algorithm is required only when the leader changes, and hence can be skipped when p_1 is the leader. Further, consider the last two steps in a round: processes sending acknowledgment to the leader (in reply to the *write*) and the subsequent *decision* message sent by the leader. These two steps can be merged into a single step in a decentralized scheme: processes send the acknowledgment to all processes; on receiving acknowledgment from a majority, a process decides immediately. These two optimizations result in the $cs_{F,DPC}$ values presented in Table 1.

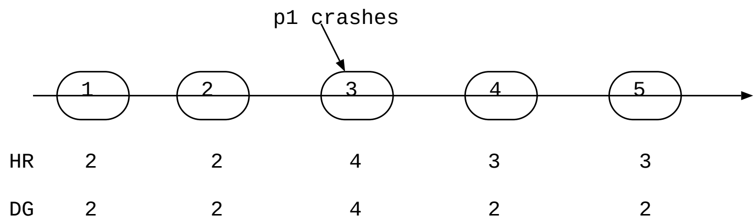

Figure 4: Repeated consensus performance

and PC algorithms are centralized and have a message complexity of $O(n)$ no matter how the processes are connected.

5.4 Repeated consensus and zero degradation

In practice, the zero degradation property of a consensus algorithm is important in case of repeated consensus based applications. Consider the [CT96] atomic broadcast algorithm, implemented as a sequence of consensus instances. Further, consider a nice run of the algorithm. If the HR consensus algorithm is used as an underlying consensus module (which is the most efficient $\diamond\mathcal{S}$-based consensus algorithm we knew of), then each consensus instance takes two communication steps. If $DG_{\diamond\mathcal{S}}$ consensus algorithm is used instead, still each consensus instance requires two communication steps.

Now, consider a slightly different run, depicted in Figure 4: process p_1 crashes during the third consensus instance (there are no other failures and the failure detector at all processes suspects p_1 after the third consensus instance). The performance of both consensus algorithms (HR and $DG_{\diamond\mathcal{S}}$) are the same for the first three consensus instances.[8] In the $DG_{\diamond\mathcal{S}}$ consensus algorithm, even though the crash of process p_1 slows the third consensus instance, other consensus instances are not affected: all subsequent consensus instances still take two communication steps (zero degradation). On the other hand, even in the absence of further failures or false suspicions, every subsequent HR consensus instance takes three communication steps. In long runs of atomic broadcast, this is a significant performance overhead. Similar performance overheads are incurred whenever atomic broadcast uses consensus algorithms which are not zero degrading.

References

[ACT00] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, May 2000.

[BGMR01] F. Brasileiro, F. Greve, A. Mostefaoui, and M. Raynal. Consensus in one communication step. In *Proceedings of the 6th International Conference on Parallel Computing Technology*, pages 42–50, Novosibirsk, Russia, September 2001.

[CHT96] 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.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[8] The performance of the third instance may be different: HR may require only 3 steps depending on when exactly p_1 crashes.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gue00] R. Guerraoui. Indulgent algorithms. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC-19)*, pages 289–298, Portland, OR, July 2000.

[HR99] M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, 1999.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 5, pages 97–146. Addison-Wesley, second edition, 1993.

[KR01] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults - a tutorial. Technical Report MIT-LCS-TR-821, MIT, May 2001.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam89] L. Lamport. The part-time parliament. Technical Report 49, Systems Research Center, Digital Equipment Corp, Palo Alto, September 1989. A revised version of the paper also appeared in ACM Transaction on Computer Systems, 16(2):133-169, May 1998.

[MR01] A. Mostefaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, March 2001.

[Sch97] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

Appendix: Correctness Proofs

Correctness of DG_Ω (Figure 1)

Lemma 1: *If a process decides v then some process has sent (*DECIDE*, v) at line 15 of some round.*
Proof: Suppose by contradiction that some process p_i decides v and no process has sent (DECIDE, v) at line 15. Process p_i decides either at line 15 or at line 22 of a round. If p_i decides at line 15 then by the algorithm, p_i must have send (DECIDE, v) at line 15, a contradiction. Therefore, p_i must have decided at line 22. So, every (DECIDE, v) message which is sent until p_i decides, is sent at line 21, and there is at least one such message. This is a contradiction because every (DECIDE, v) message which is sent at line 21 requires that a distinct (DECIDE, v) message has been sent before it (line 20). $\square$

Proposition 2. *(Validity)*: *If a process decides v, then some process has proposed v.*

Proof: If a process decides v, then some process has sent (DECIDE, v) at line 15 of some round (Lemma 1). Assume that (DECIDE, v) was sent at line 15 of round r. Any decision value sent in round r must be the *newEstimate* value of some process at round r (line 14). Further, any *newEstimate* ($\neq \perp$) value at round r must be the *estimate* of some process at the beginning of round r. Thus, v must be the *estimate* of some process at the beginning of round r. To prove validity, we show: *for any round r, if v is the estimate at some process at the beginning of round r, then v was proposed by some process.* We prove the above statement by induction on round numbers.

- *Base Step*: In round 0, every process sets its *estimate* to its own proposed value at line 4.
- *Induction Hypothesis*: At the beginning of round k, if v is the *estimate* of some process, then v was proposed by some process.
- *Induction Step*: Consider round $k+1$. Every process which executes round $k+1$ must have completed round k. The *estimate* of a process at the beginning of round $k+1$ is the same as its *estimate* at the end of round k. The *estimate* of a process at the end of round k must be the *newEstimate* of some process at round k, and any *newEstimate* ($\neq \perp$) value must be the *estimate* of some process at the beginning of round k. Thus, any *estimate* value at the beginning of round $k+1$ was an *estimate* value in the beginning of round k. Applying the induction hypothesis, any *estimate* value at the beginning of round $k+1$ is a proposed value. $\square$

Proposition 3 *(Termination)*: *Every correct process eventually decides. (Every correct process eventually returns from the propose(*) invocation.)*

Proof: We prove the proposition by contradiction. Assume that some correct process never decides. If any correct process decides then it has sent a DECIDE message to all (at line 15 or at line 21) and so every correct processes eventually receives a DECIDE message and decides (recall that messages sent to a correct process is eventually delivered), contradicting our original assumption. Thus, if some correct process never decides then no correct process ever decides. Therefore, by our original assumption, no correct process decides.

If some correct process p_i never decides, then either p_i is blocked forever in a round or is executing an infinite number of rounds. We show both cases to be impossible.

Case 1: Some correct process blocks forever in a round. Let r be the smallest round in which some correct process, say p_i, blocks forever. This can only be possible at some **wait** statement in round r. There are two **wait** statements in a round, at line 8 and at line 12.

Case 1.1: Assume that p_i blocks forever at the **wait** statement of line 8. Since no correct process blocks in any lower round (by definition of r), then every correct process sends an ESTIMATE message in round r. If $leader_i$ is correct then p_i eventually receives an ESTIMATE message from $leader_i$. If $leader_i$ is faulty then eventually $leader_i \neq \Omega.trusted$. Further, p_i receive at least $\lceil \frac{n+1}{2} \rceil - 1$ other PROPOSAL messages since there are at least $\lceil \frac{n+1}{2} \rceil$ correct processes. Thus, p_i cannot block forever at line 8.

Case 1.2: Assume that p_i blocks forever at the **wait** statement of line 12. Since no correct process blocks forever at line 8 in round r, then every correct process sends a NEWESTIMATE message. As there are at least $\lceil \frac{n+1}{2} \rceil$ correct processes, p_i receives NEWESTIMATE message from $\lceil \frac{n+1}{2} \rceil$ processes. Thus, p_i cannot block forever at line 12.

Case 2: Assume that all correct processes execute an infinite number of rounds. Consider the smallest time t such that, (i) before t, every faulty process has crashed, and (ii) after t, Ω at every correct process always outputs the same correct process, p_c.[9] By the impossibility of case 1 and the assumption that every correct process executes an infinite number of rounds, there must exist a round r such that, all correct processes start round r after time t. Every correct process sends an ESTIMATE message with $leader = p_c$ and receives ESTIMATE messages from p_c and $\lceil \frac{n+1}{2} \rceil - 1$ correct processes. Correct processes adopt the *estimate* of p_c as its *newEstimate*. Every correct process sends an NEWESTIMATE message with the same *newEstimate* ($\neq \perp$) value. After receiving non-$\perp$ *newEstimate* values from a majority of processes, every correct process sends a DECIDE message and decides at line 15. □

Lemma 4: *For any round, if a process sends a* NEWESTIMATE *message with* $newEstimate = x \neq \perp$*, then all* NEWESTIMATE *messages are sent with* $newEstimate \in \{x, \perp\}$.

Proof: Suppose by contradiction that in some round r, p_i sends a NEWESTIMATE message with $newEstimate = x \neq \perp$ and another process p_j sends a NEWESTIMATE message with $newEstimate = y \notin \{x, \perp\}$. Process p_i must have received (ESTIMATE, r, x, $leader_i$) message from $leader_i$ and (ESTIMATE, r, $*$, $leader_i$) messages from $\lceil \frac{n+1}{2} \rceil - 1$ other processes. Similarly, process p_j must have received (ESTIMATE, r, y, $leader_j$) message from $leader_j$ and (ESTIMATE, r, $*$, $leader_j$) messages from $\lceil \frac{n+1}{2} \rceil - 1$ other processes. Thus, processes p_i and p_j each received ESTIMATE messages from $\lceil \frac{n+1}{2} \rceil$ processes. Since $x \neq y$, $leader_i \neq leader_j$. As two majorities always overlap, some process must have sent two different ESTIMATE messages in round r; (ESTIMATE, r, $*$, $leader_i$) and (ESTIMATE, r, $*$, $leader_j$): a contradiction. □

Lemma 5 *(Elimination)***:** *If r is the smallest round in which some* DECIDE *message was sent at line 15, and v is the decision value sent by some process in round r, then (1) every process which completes round r has $estimate = v$ at the end of round r, and (2) every* DECIDE *message sent in round r has the decision value v.*

[9] Time t exists due to the definition of faulty processes and the property of Ω.

Proof: Assume that in round r process p_i sends a DECIDE message with value v at line 15. Process p_i must have received (NEWESTIMATE, r, v) messages from $\lceil \frac{n+1}{2} \rceil$ processes. From Lemma 4, all NEWESTIMATE messages are sent with $newEstimate \in \{v, \perp\}$. We now prove (1) by contradiction.

(1) Assume process p_j completes round r with $estimate \neq v$. Process p_j must have received NEWESTIMATE messages from $\lceil \frac{n+1}{2} \rceil$ processes and hence received at least one (NEWESTIMATE, r, v) message. Since p_j did not adopt v in line 14 or line 17, p_j must have received a NEWESTIMATE message with $newEstimate \notin \{v, \perp\}$: a contradiction.

(2) Follows directly from (1). □

Proposition 6 *(Agreement)*: *No two processes decide differently.*

Proof: If no process decides, then the proposition is trivially true. If a process decides then some process has sent a DECIDE message at line 15 of some round (Lemma 1). Let r be the smallest round in which some DECIDE message was sent at line 15 and let process p_i sends a decision value v in round r.

Assume that some process decides a value z. Some process must have sent a (DECIDE, z) message at line 15 of some round k (Lemma 1). By definition of r, $k \geq r$. If $k = r$ then by Lemma 5, $z = v$ (every DECIDE message sent at round r has the decision value v). If $k > r$ then, every process which executes round k must have completed round r. From Lemma 5, every process which completes round r has $estimate = v$ at the end round r. Therefore, no other value can be decided in any subsequent round. Thus, $z = v$. □

Proposition 7: *DG_Ω (Figure 1) solves consensus.*

Proof: Immediate from propositions 2, 3, and 6. □

Correctness of DG'_Ω (Figure 2)

Lemma 8: *If a process sends a* (DECIDE, v) *message in round 0 of Task 1, then (i) any process which starts round 1, has estimate* $= v$, *and (ii) any* (DECIDE, v') *message sent in round* 0 *has* $v' = v$.

Proof (*Sketch)*: Assume that process p_i sends a (DECIDE, v) message in round 0 of task 1. There are two cases to consider, depending on the line at which p_i sends the decision value.

Case 1: Process p_i sends (DECIDE, v) at line 13. Process p_i must have received message (ESTIMATE, 0, PV, $leader_i$) from $\lceil \frac{n+1}{2} \rceil$ processes (including the process $leader_i$) and $v =$ PV. In a given round, a process can change its *estimate* at line 17, line 21 or line 24.

Case 1.1: If a process sets its estimate at line 17 then it must have received $\lceil \frac{n+1}{2} \rceil$ PROPOSAL messages (line 15). Since, two majorities always overlap, the process must have received at least one (PROPOSAL, 0, PV, $leader_i$) message. Thus, it adopts PV as its *estimate* (line 16). In addition, if any process sends (DECIDE, x) at line 13, then $x =$ PV (since, it must have received at least one (PROPOSAL, 0, PV, $leader_i$) message).

Case 1.2: If a process adopts some value x as its *estimate* at line 21 or line 24, then some process p_j must send an NEWESTIMATE message with $newEstimate = x\ (\neq \perp)$. Process p_j must have received (ESTIMATE, 0, *, $leader_j$) from $\lceil \frac{n+1}{2} \rceil$ processes (including the process $leader_j$).

Further, as two majorities always overlap, at least one of these messages must be (ESTIMATE, 0, PV, $leader_i$). Thus, $leader_j = leader_i$ and $x =$ PV. In addition, if any process sends (DECIDE, *estimate*) at line 22, then *estimate* = PV (since, its *estimate* set to PV at line 21).

Case 2: Process p_i sends (DECIDE, v) at line 22. Process p_i must have received $\lceil \frac{n+1}{2} \rceil$ NEWESTIMATE messages with $newEstimate \neq \perp$ (including at least one NEWESTIMATE message with $newEstimate = v$). Since line 11 to line 17 do not change the *newEstimate* value at a process, we can use Lemma 4 to show that every *newEstimate* value is either v or $\perp$. So, at least $\lceil \frac{n+1}{2} \rceil$ processes must have sent $newEstimate = v$. Thus, every process which proceeds to round 1 must have received at least one (NEWESTIMATE, 0, v) message at line 23 and sets $estimate = v$. It follows that any other process which sends a DECIDE message at line 22 has decision value v. Further, from the proof of *Case 1.2*, it is obvious that if any process sends (DECIDE, PV) in line 13, then $v =$ PV. □

Proposition 9: DG'_Ω *(Figure 2) solves consensus.*

Proof *(sketch)*: The DG'_Ω algorithm differs from DG_Ω only in round 0. Validity and termination of DG'_Ω follows from the validity and termination property of DG_Ω. Agreement of DG'_Ω follows from Lemma 8 and the agreement property of DG_Ω. □