The Inherent Price of Indulgence*

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

Abstract

This paper presents a tight lower bound on the time complexity of indulgent consensus algorithms, i.e., consensus algorithms that use unreliable failure detectors. We state and prove our tight lower bound in the unifying framework of round-by-round fault detectors.

We show that any $\diamond P$-based t-resilient consensus algorithm requires at least $t+2$ rounds for a global decision even in runs that are *synchronous*. We then prove the bound to be tight by exhibiting a new $\diamond P$-based t-resilient consensus algorithm that reaches a global decision at round $t+2$ in every synchronous run. Our new algorithm is in this sense significantly faster than the most efficient indulgent algorithm we knew of (which requires $2t+2$ rounds).

We contrast our lower bound with the well-known $t+1$ round tight lower bound on consensus for the synchronous model, pointing out the price of indulgence.

1 Introduction

1.1 Context

Indulgent algorithms [7] are distributed algorithms which can tolerate unreliable failure detection [2]; i.e., algorithms where, for an arbitrary period of time, no process can distinguish a process which is up from one that has crashed: these algorithms are indulgent towards their failure detector. This characteristic makes indulgent algorithms particularly attractive in practical systems where unpredictable delays make it very hard to accurately detect failures. We consider indulgent algorithms that deterministically solve the (uniform) consensus

*This work is partially supported by the Swiss National Science Foundation (project number 510-207).

problem [5] in a message-passing distributed system with n processes: we denote by t the maximum number of processes that might fail and assume that processes can fail only by crashing.

Not surprisingly, indulgence entails a price: [2, 7] has shown that a majority of correct processes ($t < \lceil \frac{n}{2} \rceil$) is necessary for any consensus algorithm to tolerate unreliable failure detection, whereas non-indulgent algorithms can solve consensus even with a minority of correct processes. One wonders whether the unreliability of failure detection makes indulgent consensus algorithms also inherently less efficient than non-indulgent consensus algorithms. Basically, in runs where the system is synchronous (and hence the failure detection is reliable), do indulgent solutions to consensus "take longer" than non-indulgent solutions? Investigating synchronous runs of indulgent consensus algorithms is interesting because, in practice, most runs are actually synchronous.

In this paper, we address this question by comparing (1) consensus algorithms devised with a synchronous model in mind (non-indulgent algorithms) with (2) consensus algorithms devised with the unreliable failure detector $\diamond P$ in mind (indulgent algorithms).[1]

To address this question, we consider the generic round-by-round fault detector (RRFD) computation framework of [6]. Roughly speaking, in each round of that framework, every process is supposed to send messages to all processes, receive messages which are sent in that round, update its internal state depending on the messages received, and then move to the next round. While waiting for messages, a process consults the local RRFD module which outputs a set of crashed processes (some or all of these might actually be correct). A concrete RRFD model is characterized by the predicate on its RRFD, and this predicate expresses the synchrony and resilience guarantees of the model. Assumptions of the synchronous model or assumptions of $\diamond P$ are captured through concrete RRFD models, which we denote by RF_{SR} and $RF_{\diamond P}$, respectively.[2]

1.2 Background

We say that a run in an RRFD model is *synchronous* iff the RRFD also satisfies the predicates of RF_{SR} in that run. By definition, all runs in RF_{SR} are synchronous. A run of a consensus algorithm achieves a *global decision* at round k if (1) all processes which ever decide in that run, decide at round k or at a lower round and (2) at least one process decides at round k. As a measure of time complexity in a model M (RF_{SR} or $RF_{\diamond P}$), we seek the *tight lower bound* k_M such that: (1) every consensus algorithm in M has a synchronous run which requires at least k_M rounds for a global decision (i.e., every consensus algorithm

[1] Failure detector $\diamond P$ (Eventually Perfect) outputs a set of *suspected* processes at each process such that (1) (*strong completeness*) eventually every process that crashes is permanently suspected by every correct process, and (2) (*eventual strong accuracy*) there is a time after which correct processes are not suspected by any correct process. $\diamond P$ is unreliable: even if a process p_i is up at some time τ, failure detector module at some process p_j can falsely suspect p_i at τ.

[2] We give the RRFD definitions precisely in Section 2 before stating our results.

in M has a synchronous run in which some process decides at round k_M or at a higher round), and (2) there is a consensus algorithm in M which achieves a global decision at round k_M in every synchronous run.

It is well-known that $k_{RF_{SR}} = t+1$: (1) every consensus algorithm in RF_{SR} has a run which requires $t+1$ rounds for a global decision (provided $t+1 < n$) [10], and (2) a simple modification of *FloodSet* algorithm in [10] solves consensus in RF_{SR} and achieves global decision at round $t+1$ in every run. In this paper we seek $k_{RF_{\diamond P}}$: the tight lower bound for $RF_{\diamond P}$. Interestingly, the authors of [4] speculated that such a bound would be greater than $t+1$. In fact, the most efficient algorithm we knew of has a bound of $2t+2$ [8].

1.3 Contributions

The contribution of this paper is to show that $k_{RF_{\diamond P}} = t+2$; i.e., the price of indulgence is exactly "one" round.

We first show that, for every consensus algorithm A in $RF_{\diamond P}$, among all synchronous runs of A, there is at least one run in which some process decides at round $t+2$ or at a higher round, provided $0 < t < \lceil \frac{n}{2} \rceil$.[3] Our proof extends the technique of [1], used to prove the $t+1$ round lower bound for consensus algorithms in a synchronous model, to models with unreliable RRFD: indistinguishability of runs in our proof results from process crashes as well as from false suspicions. (Although we show the lower bound in the context of the uniform consensus problem, it immediately extends to the non-uniform version of the problem: [7] has shown that any indulgent algorithm which solves non-uniform consensus, also solves uniform consensus.)

Then we exhibit a consensus algorithm in $RF_{\diamond P}$ which achieves a global decision at round $t+2$ in every synchronous run. It is a flooding algorithm which tries to detect false suspicions by exchanging the set of suspected processes and expedites decision whenever it detects the absence of false suspicions.

For pedagogical reasons, we first give a "simple" version of the algorithm to show that our lower bound is tight. We then briefly explain (1) how to optimize our algorithm to achieve the time complexity lower bound for failure-free case [9]; i.e., to reach a global decision at round 2 in failure-free synchronous runs (nice runs), and (2) how to modify our algorithm to rely on a $\diamond S$-based asynchronous round model instead of $RF_{\diamond P}$.[4] The resulting algorithm is significantly more efficient (in worst-case synchronous runs, i.e., synchronous runs of the algorithm which require highest number of rounds for a global decision) than any other $\diamond S$-based consensus algorithms we know of. Our $\diamond S$-based algorithm achieves a global decision at round 2 in failure-free synchronous runs and at round $t+2$ in

[3] We exclude the following two cases. (1) $t = 0$: processes can decide after exchanging proposal values in the very first round (say on the proposal value of p_1). (2) $t \geq \lceil \frac{n}{2} \rceil$: as we have already pointed out, there is no indulgent solution to consensus when a majority of processes may fail.

[4] Failure detector $\diamond S$ (Eventually Strong) differs from $\diamond P$ in its accuracy property: $\diamond S$ ensures only (*eventual weak accuracy*) that there is a time after which some correct process is never suspected by any correct process.

```
at each process p_i
k ← 1
forever do
  compute m(i, k)
  ∀p_j ∈ Π, send m(i, k) to p_j
  wait until ∀p_j ∈ Π
    received m(j, k) or p_j ∈ D(i, k)
  k ← k + 1
```

Figure 1: An abstract RRFD algorithm

every other synchronous runs. In contrast, the $\diamond S$-based consensus algorithm of [8], which used to be the most efficient in worst-case synchronous runs among the indulgent consensus algorithms we knew of, has a synchronous run which requires $2t + 2$ rounds for a global decision.

1.4 Roadmap

Section 2 briefly describes the distributed system model in which we state and prove our result. Section 3 formally states our lower bound result with an intuitive proof for a simple, yet non-trivial, case. The detailed proof of the result is given in Appendix A. Section 4 exhibits a consensus algorithm which achieves the lower bound. Its correctness proof is given in Appendix B. We also detail the optimization of our algorithm for failure-free synchronous runs in Appendix C.

2 Model

We consider a crash-stop message-passing distributed system consisting of a set of $n > 2$ processes: $\Pi = \{p_1, p_2, ..., p_n\}$. Every pair of processes can communicate through *send* and *receive* primitives, which emulate a *reliable* communication channel in the following sense: (1) each message sent from a correct process to a correct process is eventually received, (2) each message is received at most once, and (3) the channel does not create or alter any message. A process executes the deterministic algorithm assigned to it or *crashes*. Processes do not recover from a crash. A *correct* process is a process that never crashes; all other processes are *faulty*.

A *run* of an RRFD based distributed algorithm [6] proceeds in rounds with processes moving from one round to the next higher round until the algorithm terminates. In each round k, every process p_i is supposed to execute the following steps: (1) p_i computes the message for this round, $m(i, k)$, (2) p_i sends $m(i, k)$ to all processes, and (3) p_i receives some of the messages sent at round k. While executing the third step, the processes consult the RRFD. For a given round k, the RRFD outputs at every process p_i a set of possibly faulty processes $D(i, k)$, such that p_i receives $m(*, k)$ at round k from every processes in $\Pi - D(i, k)$. An abstract RRFD based algorithm is described in Figure 1.

An RRFD can be unreliable, namely, indicate a process to be faulty when it is actually up. A concrete RRFD model can be completely defined by predicates on the set $D(i,k)$. We say that a process p_i *suspects* p_j when p_j is in the set of suspected processes output by RRFD at p_i. It is worth noticing that a round is "communication closed", i.e., for any message m, either m is received by a process p_i in the same round in which it is sent, or m is never received by p_i. The restriction of a run r of an algorithm A to the first k rounds is called a *k-round partial run* and is denoted by r_k. (For each process p_i, r_k contains all steps of p_i in r until p_i either crashes or p_i completes round k.)

Synchronous round model: The RRFD model RF_{SR}, where at most t processes can fail by crashing, is defined by the following two predicates on $D(i,k)$ [6] ($\mathbb{N}$ denotes the set of positive integers):

$A1$. $((\forall k \in \mathbb{N})(\forall p_i \in \Pi)(p_i \notin D(i,k))) \;\wedge\; (|\cup_{k\in\mathbb{N}} \cup_{p_i\in\Pi} D(i,k)| \leq t)$

$A2$. $(\forall k \in \mathbb{N})(\forall p_l \in \Pi)(\cup_{p_i\in\Pi} D(i,k) \subseteq D(l,k+1))$

Roughly speaking, predicate $A1$ states that in any given run, a process never suspects itself, and no more than t distinct processes are ever suspected. $A2$ states that if a processes p_j crashes in round k, no processes receives a messages from p_j in a higher round.

Asynchronous round model enriched with $\diamond P$: We define the RRFD model $RF_{\diamond P}$, where at most t processes can fail by crashing, by the following three predicates on $D(i,k)$:[5]

$B1$. $(\forall k \in \mathbb{N})(\forall p_i \in \Pi)(|D(i,k)| \leq t)$

$B2$. $(\exists k' \in \mathbb{N})(((\forall k \geq k')(\forall p_i \in \Pi)(p_i \notin D(i,k))) \;\wedge\; (|\cup_{k\geq k'} \cup_{p_i\in\Pi} D(i,k)| \leq t))$

$B3$. $(\exists k' \in \mathbb{N})(\forall k \geq k')(\forall p_l \in \Pi)(\cup_{p_i\in\Pi} D(i,k) \subseteq D(l,k+1))$

Roughly speaking, $B1$ expresses the resilience guarantee of the model: at every round k, a process eventually receives round k messages from at least $n-t$ processes. Predicates $B2$ and $B3$ simply state that the $RF_{\diamond P}$ eventually provides synchronous guarantees.

Synchronous run in $RF_{\diamond P}$: We say that a run r of an algorithm in $RF_{\diamond P}$ is synchronous iff the RRFD satisfies predicates $A1$ and $A2$ in r.

[5] Note that we give here an RRFD model with slightly stronger synchrony properties than what $\diamond P$ actually ensures: eventually, $RF_{\diamond P}$ provides similar guarantees as RF_{SR}. This strengthens our lower bound result: if achieving a global decision at round $t+1$ in every synchronous run is impossible in $RF_{\diamond P}$ then obviously it is impossible with weaker assumption of $\diamond P$. After describing our consensus algorithm in $RF_{\diamond P}$, we then show how to modify the algorithm to rely on asynchronous round model with $\diamond S$.

A consensus algorithm assists a set of processes to decide on a single value among the values proposed by the processes. We define consensus here using two primitives: propose(v) and decide(v). Each process proposes a value v through the function propose(v) and a process decides v through decide(v). Consensus ensures the following properties: (i) (*validity*) if a process decides v then some process has proposed v, (ii) (*uniform agreement*) no two processes decide differently, (iii) (*termination*) every correct process eventually decides, and (iv) (*integrity*) no process decides twice.

An RRFD-based consensus algorithm A at each process p_i is invoked through procedure propose($*$) and progresses as a sequence of an arbitrarily large number of RRFD-based rounds until either the consensus properties are satisfied or p_i crashes.

3 Lower Bound

Proposition 1. *Every consensus algorithm in $RF_{\diamond P}$, with $0 < t < \lceil \frac{n}{2} \rceil$, has a synchronous run in which some process decides at round $t + 2$ or at a higher round.*

3.1 Proof overview

The basic structure of the proof is as follows. We assume for a contradiction that there is a consensus algorithm A in $RF_{\diamond P}$ which achieves a global decision at round $t + 1$ in every synchronous run. Then we construct two $(t + 1)$-round partial runs r and r' of A with the following properties:

(1) $t - 1$ processes crash in first t round of r and r'

(2) except some process p_i, no other process can distinguish r from r'

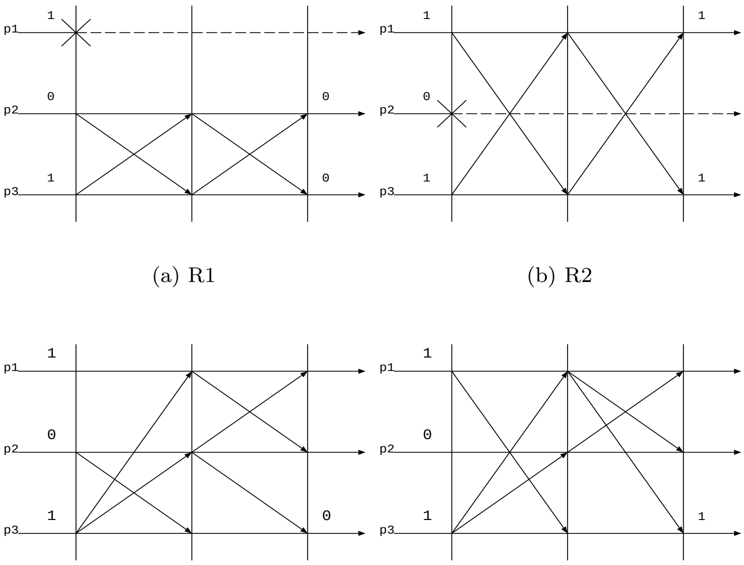
(3) r and r' appear as synchronous runs to p_i

(4) p_i decides different values and then crashes at the end of r and r'

Roughly speaking, since the processes (other than p_i) cannot distinguish r from r', in any extension of r (or r'), these processes can never learn the decision value of p_i. The construction of the first $t - 1$ rounds of r and r' follows the bivalency-based forward induction on round numbers, introduced in [1]. (However, our notion of bivalency is different.) For the construction of the next two rounds, we use process crashes as well as false suspicions to generate the required indistinguishability. The complete proof (providing the detailed construction of the runs) is presented in Appendix A. In the following, we illustrate the idea of the proof for a simple, yet non-trivial, case.

3.2 A specific case

We informally explain here why there cannot exist any consensus algorithm A in $RF_{\diamond P}$, with $\Pi = \{p_1, p_2, p_3\}$ and $t = 1$, such that, in every synchronous run of A, a global decision is achieved within round 2.



(a) R1 (b) R2

(c) R3 (d) R4

Figure 2: Consensus runs

Assume for a contradiction that there exists a binary consensus algorithm A such that, in every synchronous run of A, no process decides after round 2. Without loss of generality, we can assume that, in every synchronous run of A, the processes decide exactly at the end of round 2. Remember that (1) runs with false suspicions are necessarily non-synchronous, and (2) property $B1$ of $RF_{\diamond P}$ requires that, in any run of A, a process can suspect at most one process at a time (because $t = 1$).

We construct two synchronous runs of A, R_1 and R_2, and two partial runs of A, R_3 and R_4. R_3 and R_4 are 2-round non-synchronous partial runs. In each case, p_1 proposes 1, p_2 proposes 0, and p_3 proposes 1. The first two rounds of each run are depicted in Figure 2.[6]

- R_1: Process p_1 crashes initially. No other process crashes and there is no false suspicion. Without loss of generality, we assume the decision value to be 0,[7] i.e, p_2 and p_3 decide 0 at the end of round 2. (Recall that in synchronous runs of A, correct processes decide at the end of round 2.)

[6] The following two types of messages are not shown in the Figure 2 for clarity: (1) messages sent by a process to itself, and (2) messages "lost" due to false suspicion. The presence of messages lost due to false suspicion is evident in each run (remember that, in every round, every process which is up sends messages to all other processes), e.g., in the first round of R_3, p_1 sends messages to p_2 and p_3 but neither of the messages is received because p_2 and p_3 (falsely) suspect p_1.

[7] Notice that the decision value in R_1 does not depend on the value proposed by p_1. There-

- R_2: Process p_2 crashes initially. No other process crashes and there is no false suspicion. Clearly, the decision value in R_2 should be the same if p_2 had proposed 1 instead of 0. Hence, (by consensus validity) the decision value is 1, i.e., p_1 and p_3 decide 1 at the end of round 2.

- R_3: None of the processes crash. In round 1, p_2 and p_3 falsely suspect p_1, and p_1 falsely suspects p_2. In round 2, p_1 and p_2 falsely suspect p_3, and p_3 falsely suspects p_1. Process p_3 decides at the end of round 2 because p_3 cannot distinguish the first two rounds of R_1 from R_3. To see why, notice that in both cases, p_3 receives no message from p_1. Obviously, p_2 sends identical messages to p_3 at round 1 of R_1 and at round 1 of R_3. Furthermore, p_2 can only distinguish the runs at the end of round 2 (when p_2 receives a message from p_1 and suspects p_3). Hence, p_2 sends identical messages to p_3 at round 2 of R_1 and at round 2 of R_3. Thus, p_3 receives identical messages in both runs. Consequently, in every extension of R_3, (i) (as in R_1) p_3 decides 0 at the end of round 2, and (ii) (by consensus agreement) in any extension of R_3, p_1 and p_2 eventually decide 0.

- R_4: None of the processes crash. In round 1, p_1 and p_3 falsely suspect p_2, and p_2 falsely suspect p_1. In round 2, p_1 and p_2 falsely suspect p_3, and p_3 falsely suspects p_2. Process p_3 decides at the end of round 2 because p_3 cannot distinguish the first two rounds of R_2 from R_4. Thus, in every extension of R_4, (i) (as in R_2) p_3 decides 1 at the end of round 2, and (ii) (by consensus agreement) in any extension of R_4, p_1 and p_2 eventually decide 1.

Notice that p_1 and p_2 cannot distinguish R_3 from R_4. Each process receives identical messages in both partial runs. Consider any run R_5 which extends R_3 such that p_3 crashes at round 3 before sending any message. In R_5, p_1 and p_2 decide 0 (by consensus agreement). Now replace the first two rounds of R_5 by R_4. Since, p_1 and p_2 cannot distinguish R_3 from R_4, they still decide 0 in R_5: violating consensus agreement, as p_3 decides 1 in R_4.[8]

4 The consensus algorithm

We present here a consensus algorithm in $RF_{\diamond P}$, which we denote by A_{t+2}, for $0 < t < \lceil \frac{n}{2} \rceil$. A_{t+2} achieves the lower bound of Proposition 1. Namely, besides solving consensus, A_{t+2} satisfies the following property:

Fast Decision: In every synchronous run of A_{t+2}, any process which ever decides, decides at round $t+2$ or at a lower round.

---

fore, if the decision value is 1, we can easily modify the proof by constructing runs in which p_1 proposes 0.

[8]Notice that partial runs R_3 and R_4, and process p_3 respectively correspond to the r, r', and p_i of Section 3.1.

The algorithm assumes an underlying independent consensus module C,[9] accessed by procedures propose$_C(*)$ and decide$(*)$. The fast decision property is achieved by A_{t+2} regardless of the time complexity of C. More precisely, our algorithm assumes:

(1) the RRFD computation model $RF_{\diamond P}$ with $0 < t < \lceil \frac{n}{2} \rceil$

(2) no process ever suspects itself

(3) an independent consensus algorithm C in $RF_{\diamond P}$

(4) the set of proposal values in a run is a totally ordered set; e.g., each process p_i can tag its proposal value with its index i and then the values can be ordered based on this tag

For presentation simplicity, we consider a slightly different consensus integrity property: for every process p_i, no two decide$(*)$ invocations at p_i have different values. Thus, even though we allow each process to decide more than once, the decision value should not change between decisions. The original integrity property can be recovered by a procedure which accepts the first decision value and ignores the rest.

4.1 Basic idea

Our algorithm is a variant of the *FloodSetWS*[10] algorithm of [3], modified for exchanging and tracking false suspicions. The algorithm has two phases: Phase 1 lasts the first $t + 1$ rounds and Phase 2 involves round $t + 2$ and the underlying consensus algorithm C. In Phase 1, processes exchange their estimates of the decision (initialized to the proposal value) and every process updates its estimate to the minimum of all estimates seen in the round. The primary objective of repeating this exchange for $t + 1$ rounds is to converge towards the same estimate at all processes. However, this may be hindered by false suspicions, i.e., processes may have different estimates at the end of Phase 1. Therefore, the algorithm tries to detect the false suspicions to ensure the following *elimination* property: given any two processes which complete Phase 1, either both processes have the same estimate values or at least one of them detects a false suspicion. The algorithm does not try to detect all false suspicions but only those which can result in different estimate values at the end of Phase 1.

At round $t + 2$ (Phase 2), the processes exchange their (new) estimates: if a process detects a false suspicion, then its new estimate is set to $\perp$; otherwise, the new estimate is the estimate value at the end of Phase 1. Due to the elimination property of Phase 1, in every run, the number of distinct new estimate values different from $\perp$ is at most one. Processes decide at round $t + 2$ only if at least $n - t$ processes send non-$\perp$ estimate value. Otherwise, achieving decision is delegated to algorithm C. (Due to consensus termination property of C, at every correct process, procedure propose$_C(*)$ eventually invokes decide$(*)$.)

[9] This algorithm can be any traditional $\diamond P$-based or $\diamond S$-based consensus algorithm (e.g., the one based on $\diamond S$ in [2]) transposed to the $RF_{\diamond P}$ model.

[10] Consensus algorithm FloodSetWS assumes *perfect* failure detection (P) and achieves global decision at round $t + 1$ in every run. It is itself inspired by the FloodSet consensus algorithm of [10] in a synchronous system.

at each process p_i
procedure propose(v_i)
  $k_i \leftarrow 1$
  Phase 1
  while $k_i \leq t + 1$
    compute()
    $\forall p_j \in \Pi$, send(ESTIMATE, k_i, est_i, $Halt_i$) to p_j
    wait until $\forall p_j \in \Pi$, received(ESTIMATE, k_i, $*$, $*$) from p_j **or** $p_j \in D(i, k_i)$
    $k_i \leftarrow k_i + 1$
  Phase 2
  compute()
  $\forall p_j \in \Pi$, send(NEWESTIMATE, nE_i) to p_j
  wait until $\forall p_j \in \Pi$, received(NEWESTIMATE, $*$) from p_j **or** $p_j \in D(i, k_i)$
  if every received(NEWESTIMATE, nE) has $nE \neq \perp$ **then**
    $vc_i \leftarrow$ any one of the nE values received
    decide(vc_i)
  else if received any (NEWESTIMATE, nE') message *s.t.* $nE' \neq \perp$ **then**
    $vc_i \leftarrow nE'$
  propose$_C$(vc_i)

procedure compute()
 if $k_i = 1$ **then**
   $msgSet_i \leftarrow \emptyset$; $mistake_i \leftarrow false$; $est_i \leftarrow v_i$; $Halt_i \leftarrow \emptyset$; $nE_i \leftarrow v_i$; $vc_i \leftarrow v_i$
 if $2 \leq k_i \leq t + 2$ **then**
   $msgSet_i \leftarrow \{$(ESTIMATE, $k_i - 1$, $*$, $Halt_j$) $\mid$ p_i received(ESTIMATE, $k_i - 1$, $*$, $Halt_j$) from p_j **and** $p_j \notin Halt_i\}$
   $Halt_i \leftarrow Halt_i \cup \{p_j \mid p_i$ has not received(ESTIMATE, $k_i - 1$, $*$, $*$) from $p_j\}$
   if p_i received(ESTIMATE, $k_i - 1$, $*$, $Halt_j$) from some process p_j *s.t.* $p_i \in Halt_j$ **then**
     $mistake_i \leftarrow true$
   $est_i \leftarrow$ **Min**$\{est \mid$ (ESTIMATE, $k_i - 1$, est, $*$) $\in msgSet_i\}$
 if $k_i = t + 2$ **then**
   if $\mid Halt_i \mid > t$ **or** $mistake_i = true$ **then**
     $nE_i \leftarrow \perp$
   else
     $nE_i \leftarrow est_i$

Figure 3: The consensus algorithm

4.2 Description (Figure 3)

Processes invoke propose($*$) with their respective proposal value, and the procedure progresses in RRFD based rounds. After receiving messages in any round k (in Phase 1), the processes invoke procedure compute() at the beginning of round $k+1$ to update their local states. The algorithm tries to achieve consensus in the first $t+2$ rounds. Irrespective of whether a process decides at round $t+2$ or not, the process invokes the underlying consensus algorithm C.

Every process p_i maintains the following variables: (1) k_i: the current round number; (2) est_i: the estimate of p_i which is set to the minimum value seen by p_i till round $k_i - 1$, initialized to the proposal value v_i; (3) $Halt_i$: the set of processes suspected by p_i in any lower round, (4) nE_i: the new estimate of p_i, and (5) vc_i: the proposal value for the underlying consensus algorithm C, initialized to the proposal value v_i.

Phase 1: In this phase, which consists of the first $t+1$ rounds, processes exchange ESTIMATE messages containing est and $Halt$. On receiving the messages at round k, p_i updates its variables at the beginning of round $k+1$ (by invoking the procedure compute()) as follows:

- $msgSet_i$ is the set of messages received by p_i at round k such that p_i did not suspect the sender in some round lower than k (i.e., once p_i suspects a process p_j, all subsequent messages from p_j are ignored by p_i while computing $msgSet_i$).

- est_i is updated as the minimum est value in $msgSet_i$.

- $Halt_i$ is the set of processes suspected by p_i at round k or some lower round.

- $mistake_i$ is $true$ iff p_i detects that some process has falsely suspected p_i. Namely, if p_i receives a message from any process p_j such that $p_i \in Halt_j$, then p_i sets $mistake_i$ as $true$.

Phase 2: This phase starts at round $t+2$. At round $t+2$, processes exchange their nE (NEWESTIMATE messages) and these are adopted as follows. If p_i does not detect a false suspicion within the first $t+1$ rounds, then it sets nE_i to the minimum est value it has seen (i.e., the latest est_i value). Otherwise, nE_i is set to $\perp$. Process p_i detects a false suspicion when (line 29) the cardinality of $Halt_i$ is greater than t (p_i has suspected more than t processes, therefore at least one of the suspicions is false) or $mistake_i$ is $true$ (some process falsely suspected p_i). On exchanging nE values, if p_i receives only non-$\perp$ nE values, then p_i decides immediately on any nE value received and sets vc_i to that value. Otherwise, either p_i receives some $nE' \neq \perp$ and sets vc_i to nE', or every nE value received by p_i is $\perp$ and vc_i retains its initial value, v_i. Subsequently, p_i invokes propose$_C(vc_i)$.

4.3 Outline of the proof

The validity and termination properties of the algorithm are rather straightforward. The integrity and agreement property follows from our elimination property: if there are two distinct processes p_i and p_j such that, p_i and p_j

7: **wait until** ($\forall p_j \in \Pi$, received(ESTIMATE, k_i, $*$, $*$) from p_j or $p_j \in \Diamond S_{p_i}$) **and** (received(ESTIMATE, k_i, $*$, $*$) from at least $n-t$ processes)
12: **wait until** ($\forall p_j \in \Pi$, received(NEWESTIMATE, $*$) from p_j or $p_j \in \Diamond S_{p_i}$) **and** (received(NEWESTIMATE, $*$) from at least $n-t$ processes)

Figure 4: Modifications for using $\Diamond S$

send NEWESTIMATE messages with $nE_i \neq \perp$ and $nE_j \neq \perp$, respectively, then $nE_i = nE_j$. It immediately follows that if any process decides on some value d at round $t+2$, then every process which completes round $t+2$ has invoked propose$_C(d)$. A detailed correctness proof of the elimination property of the algorithm is given in Appendix B. Integrity and agreement properties follows from the agreement and validity property of C.

To see how the fast decision property is ensured, notice that the condition at line 29 is *false* in every synchronous run: (1) From predicate $A1$ it follows that no process can suspect more than t processes in any synchronous run. Thus, the size of the set $Halt$ is never greater than t in a synchronous run. (2) Consider process p_i. Variable $mistake_i$ is set to *true* in some round k only if p_i received a message from some process p_j such that $Halt_j$ contains p_i. So, p_j must have suspected p_i at some round $k' < k$. As p_i is up at round k, predicate $A1$ and $A2$ is violated ($p_i \in D(j, k')$ but $p_i \notin D(i, k)$), and hence, in synchronous runs $mistake_i$ is always *false*.

Hence, in every synchronous run, processes set nE different from $\perp$. Thus, every NEWESTIMATE message has $nE \neq \perp$, and no process completes round $t+2$ without deciding (line 13).

4.4 Extensions

1. Algorithm A_{t+2} can be easily transformed to a consensus algorithm with $\Diamond S$ [2, 8], which we denote by $A_{\Diamond S}$, as follows: (1) substitute underlying consensus algorithm C by any $\Diamond S$-based consensus algorithm C' (e.g., of [2]), and (2) modify line 7 and line 12 as shown in Figure 4. Correctness of $A_{\Diamond S}$ is easy to verify, since consensus termination is ensured by the presence of at least $n-t$ correct processes, and the termination property of C'. More interestingly, $A_{\Diamond S}$ retains the fast decision property of A_{t+2} because this property is relevant only in synchronous runs where the synchrony guarantees are much stronger than those of either $RF_{\Diamond P}$ or $\Diamond S$-based asynchronous rounds.

2. Algorithm A_{t+2} (and $A_{\Diamond S}$) can be easily optimized to achieve a global decision at round 2 in failure-free synchronous runs as follows. If a process detects absence of suspicion at round 1 (i.e., received $Halt = \emptyset$ from n processes at round 2) then it can safely conclude that the estimates at all processes at the end of round 1 are identical and equal to the minimum value among all proposed values. Thus, the process can decide on any estimate it receives at round 2. Appendix C details this optimization and sketches its correctness proof.

5 Acknowledgment

We thank Petr Kouznetsov, Bastian Pochon, and the anonymous reviewers for their helpful comments on earlier drafts of the paper.

References

[1] M. K. Aguilera and S. Toueg. A simple bivalency proof that t-resilient consensus requires $t+1$ rounds. *Information Processing Letters*, 71(3-4):155–158, 1999.

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

[3] B. Charron-Bost, R. Guerraoui, and A. Schiper. Synchronous system and perfect failure detector: solvability and efficiency issues. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 523–532, New York, June 2000.

[4] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[5] 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.

[6] E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC-17)*, pages 143–152, Puerto Vallarta, Mexico, 1998.

[7] 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.

[8] 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.

[9] 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.

[10] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

A Proof of Proposition 1

Proposition 1. *Every consensus algorithm in $RF_{\diamond P}$ with $0 < t < \lceil \frac{n}{2} \rceil$ has a synchronous run in which some process decides at round $t+2$ or at a higher round.*

Proof: Suppose by contradiction that there is a binary consensus algorithm A (possible proposal values are 0 and 1) in every synchronous run of which, any process which ever decides, decides at the end of round $t+1$. We prove five lemmata (Lemma 2 to Lemma 5) on algorithm A. Lemma 5 contradicts Lemma 2.

Before proving the lemmata we propose some definitions and notations. A synchronous run r of A is a *serial* run iff at most one process may crash in every round of r. Since every serial run is a synchronous run, in every serial run of A, every process which ever decides, decides at the end of round $t+1$. A finite execution of A is an *l-round serial partial run* iff it is a restriction of some serial run of A to the first l rounds. A m-round partial run r_m is a *serial extension* of an l-round serial partial run r_l $(l < m)$ iff (1) r_l is the restriction of r_m to the first l rounds, and (2) r_m is a m-round serial partial run. A m-round partial run r_m is an *asynchronous extension* of a l-round serial partial run r_l $(l < m)$ iff (1) r_l is the restriction of r_m to the first l rounds, and (2) $(l < k \leq m)(\forall p_x \in \Pi)(\cup_{p_i \in \Pi} D(i,l) \subseteq D(x,k))$.[11]

A k-round serial partial run r_k is 0-valent (1-valent) iff the only decision value in all serial extension of r_k is 0 (1). A k-round serial partial run is univalent if it is either 0-valent or 1-valent; otherwise, it is bivalent. An initial configuration C_0 is 0-valent (1-valent) iff the only possible decision value in all serial runs starting from C_0 is 0 (1). An initial configuration is univalent if it is either 0-valent or 1-valent; otherwise, C_0 is bivalent.

We denote the message sent by any process p_i at round k of run r by $m_r(i,k)$. $M_r(i,k)$ denotes the set of messages received by p_i at round k of run r.

Lemma 2: *Every t-round serial partial run is univalent.*

Proof: Suppose by contradiction that there is a t-round serial partial run r_t which is bivalent. Suppose that r^0 is a serial extension of r_t such that no process crashes after round t. Without loss of generality we assume that r^0 has decision value 0. Since run r^0 is serial, every processes which ever decides in r^0, decides 0 at the end of round $t+1$. Furthermore, as r_t is bivalent, there is a serial run r^1 which has decision value 1: every process which ever decides in r^1, decides 1 at the end of round $t+1$. Notice that as both runs r^0 and r^1 are extensions of r_t, processes cannot distinguish the runs at the beginning of round $t+1$, and therefore, the messages sent by any process at round $t+1$ are identical in both runs, i.e., $\forall p_l \in \Pi$, $m_{r^0}(l,t+1) = m_{r^1}(l,t+1)$.

[11] Note that (1) any serial extension is also an asynchronous extension, (2) in asynchronous extensions which are not serial, processes may be falsely suspected, and (3) condition 2 in the definition of asynchronous extensions states that, if a process is suspected in a serial partial run r_l then it continues to be suspected in every asynchronous extension of r_l.

Consider a process p_i which is correct in both runs r^0 and r^1 ($t < \lceil \frac{n}{2} \rceil$ implies that there is a process which is correct in both runs). $M_{r^0}(i, t+1)$ and $M_{r^1}(i, t+1)$ are the set of messages received by p_i at round $t+1$ of r^0 and r^1, respectively. Since p_i is correct, it must decide (at round $t+1$ of serial runs r^0 and r^1). To decide at round $t+1$, p_i must be able to distinguish r^0 from r^1 at round $t+1$, which implies that $M_{r^0}(i, t+1) \neq M_{r^1}(i, t+1)$. As no process crashes at round $t+1$ of r^0, $M_{r^1}(i, t+1) \subset M_{r^0}(i, t+1)$.

Now consider an asynchronous extension of r_t by one round, $a^{0,1}$. Round $t+1$ of $a^{0,1}$ is identical to round $t+1$ of r^0 except that p_i receives $M_{r^1}(i, t+1)$ instead of $M_{r^0}(i, t+1)$ (recall that $M_{r^1}(i, t+1) \subset M_{r^0}(i, t+1)$), i.e., p_i is the only process which can distinguish the first $t+1$ rounds of r^0 from the partial run $a^{0,1}$. Process p_i cannot distinguish the partial run $a^{0,1}$ from the first $t+1$ rounds of r^1 and decides 1 at the end of $a^{0,1}$. Consider a process p_j which is correct in r^0 and distinct from p_i ($0 < t < \lceil \frac{n}{2} \rceil$ implies that $t+2 \leq n$, i.e., there are two correct processes in any run). Clearly, p_j cannot distinguish the first $t+1$ rounds of r^0 from $a^{0,1}$. Thus, p_j decides 0 in $a^{0,1}$, and any extension of $a^{0,1}$ violates consensus agreement: a contradiction. $\Box$

Lemma 3: *There is an initial configuration which is bivalent.*
Proof: Suppose by contradiction that every initial configuration is univalent. Consider the initial configurations C_0 and C_n in which all processes propose 0 and 1, respectively. From consensus validity it follows that C_0 is 0-valent and C_n is 1-valent. Define C_i ($0 < i < n$) as the initial configuration in which every process p_j such that $j \leq i$ proposes 1 and all other processes propose 0. Consider a serial run r_{C_i} starting from C_i ($0 \leq i < n$) in which process p_{i+1} crashes initially and other processes decide $d \in \{0, 1\}$ at round $t+1$. Notice that even if the initial configuration in r_{C_i} is changed to C_{i+1}, the decision value remains d (because p_{i+1} crashes before sending any messages in r_{C_i}). Thus, if C_i ($0 \leq i < n$) is d-valent then C_{i+1} is also d-valent.

Using the above result and a simple induction we can show that, if C_0 is 0-valent, then so is C_n: a contradiction. $\Box$

Lemma 4: *There is a* $(t-1)$*-round serial partial run which is bivalent.*
Proof: The proof is by induction on round number k ($0 \leq k < t-1$).

Base Step: From Lemma 3 it follows that there is a 0-round serial run which is bivalent.

Induction Hypothesis: There is a k-round serial partial run r_k which is bivalent ($0 \leq k < t-1$).

Induction Step: We assume that every one round serial extension of r_k is univalent. We show that this leads to a contradiction. Therefore, there is a one round serial extension of r_k which is bivalent, and hence, there is a $(k+1)$-round serial partial run which is bivalent.

Suppose that every one round serial extension of r_k is univalent. Let r^0_{k+1} be a $(k+1)$-round serial partial run which is an extension of r_k such that no process crashes at round $k+1$. Without loss of generality, we can assume that r^0_{k+1} is 0-valent. Since r_k is bivalent, there is a $(k+1)$-round serial partial run r^*_{k+1} which is an extension of r_k and which is 1-valent. There must be exactly one process p'_1 which crashes in round $k+1$ of r^*_{k+1} and there is a (possibly empty) set of processes $\{p'_2, ..., p'_m\}$ that can distinguish r^0_{k+1} from r^*_{k+1} $(0 \leq m-1 < n)$: i.e., processes which received a message from p'_1 at round $k+1$ of r^0_{k+1} and did not receive a message from p'_1 at round $k+1$ of r^*_{k+1}.

Consider the following $(k+1)$-round serial partial runs $r^1_{k+1}, ..., r^m_{k+1}$ such that: (1) r^1_{k+1} is identical to r^0_{k+1}, except that in r^1_{k+1}, p'_1 crashes at round $k+1$, though the round $k+1$ message sent from p'_1 to other processes are received at round $k+1$. (2) r^j_{k+1} $(2 \leq j \leq m)$ is identical to r^0_{k+1} except that, in r^j_{k+1}, p'_1 crashes at round $k+1$ and does not send $(k+1)$-round messages to $\{p'_2, ..., p'_j\}$ (though p'_1 sends $(k+1)$-round messages to $\{p'_{j+1}, ..., p'_m\}$ and those messages are received in the same round). Now consider the following two claims which contradicts the fact that r^*_{k+1} is 1-valent.

4.1. If r^i_{k+1} $(0 \leq i < m)$ is 0-valent then so is r^{i+1}_{k+1}: Partial runs r^i_{k+1} and r^{i+1}_{k+1} differ only in the state of process p'_{i+1} at the end of round $k+1$. Consider a $k+2$ round serial extension r_{k+2} of r^i_{k+1} in which p'_{i+1} crashes at the beginning of round $k+2$ (before sending any message in round $k+2$) and no other processes crash in round $k+2$. Also, consider a $k+2$ round serial extension r'_{k+2} of r^{i+1}_{k+1} in which p'_{i+1} crashes at the beginning of round $k+2$ (if $p'_{i+1} = p'_1$ then it has already crashed in round $k+1$) and no other process crashes in round $k+2$.[12] Obviously, at the end of round $k+2$ no process can distinguish r_{k+2} from r'_{k+2}. Note that since $k+2 < t+1$, processes decide after round $k+2$. Hence, there are serial extensions of r^i_{k+1} and r^{i+1}_{k+1} which are indistinguishable at the end of round $t+1$. So, if r^i_{k+1} $(0 \leq i < m)$ is 0-valent, then r^{i+1}_{k+1} is also 0-valent. It follows that r^m_{k+1} is 0-valent.

4.2. r^*_{k+1} is 0-valent: Serial partial runs r^*_{k+1} and r^m_{k+1} are identical. Therefore, r^*_{k+1} is 0-valent: a contradiction. □

Lemma 5: *There is a t-round serial partial run which is bivalent.*
Proof: Suppose by contradiction that every t-round serial partial run is univalent. From Lemma 4 we know that there is a bivalent $(t-1)$-round serial partial run, which we denote by r_{t-1}. Let r^0_t be a one round serial extension of r_{t-1} such that no process crashes at round t. Without loss of generality, we can assume that r^0_t is 0-valent. Since r_{t-1} is bivalent, there must be a one round serial extension r^*_t of r_{t-1} which is 1-valent. There must be exactly one process p'_1 which crashes in round t of r^*_t and there is a (possibly empty) set of processes $\{p'_2, ..., p'_m\}$ that can distinguish r^0_t from r^*_t $(0 \leq m-1 < n)$: i.e., processes which received a message from p'_1 at round t of r^0_t and did not receive a message from p'_1 at round t of r^*_t.

Consider the following t-round serial partial runs $r^1_t, ..., r^m_t$ such that: (1) r^1_t is identical to r^0_t, except that in r^1_t, p'_1 crashes at round t, though the round t message sent from p'_1 to other processes are received at round t. (2) r^j_t $(2 \leq$

[12] Note that, p'_{i+1} can crash at the beginning of round $k+2$ in r'_{k+2} because, by the definition of serial runs, at most $k+1 < t$ processes can crash in the first $k+1$ rounds. $k+1 < t$ because the induction is done over $0 \leq k < t-1$.

$j \leq m$) is identical to r_t^0, except that in r_t^j, p'_1 crashes at round t and does not send t-round messages to $\{p'_2, ..., p'_j\}$ (though p'_1 sends t-round messages to $\{p'_{j+1}, ..., p'_m\}$ and those messages are received in the same round). Now consider the following two claims which contradicts the fact that r_t^* is 1-valent.

5.1. If r_t^i $(0 \leq i < m)$ is 0-valent then so is r_t^{i+1}: The proof is given in the following subsection. The claim implies that r_t^m is 0-valent.

5.2. r_t^* is 0-valent: Partial runs r_t^m and r_t^* are identical. Therefore r_t^* is 0-valent: a contradiction.

Proof of Claim 5.1

The proof of Claim 4.1 does not work for the present case. To see why, notice that in Claim 4.1, $k+1$ processes have crashed in serial partial run r_{k+1}^{i+1}. Since $k+1 < t$ (in Lemma 4), we can crash one more process in any extension of r_{k+1}^{i+1}, which is necessary to show that r_{k+1}^i and r_{k+1}^{i+1} have the same valency. However, in the present case, t processes have already crashed in r_t^{i+1}.

Proof: Suppose by contradiction that r_t^i is 0-valent and r_t^{i+1} is 1-valent. Serial partial runs r_t^i and r_t^{i+1} differ only in the state of p'_{i+1} at the end of round t. There are two cases: (1) $p'_{i+1} = p'_1$, or (2) $p'_{i+1} \neq p'_1$.

If $p'_{i+1} = p'_1$ (i.e., p'_{i+1} is up at the end of $r_t^i = r_t^0$ but crashes in $r_t^{i+1} = r_t^1$), then we reach a contradiction as follows. From the definition of serial runs we know that at most t processes can crash in r_t^{i+1}. Since r_t^i and r_t^{i+1} are identical except for state of p'_{i+1} (p'_{i+1} crashes in r_t^{i+1} but not in r_t^i), at most $t-1$ processes could have crashed in r_t^i. So, we can construct a serial run r' by extending r_t^i in which p'_{i+1} crashes at the beginning of round $t+1$ (before sending any message in that round). From round $t+1$ onwards, no process can ever learn whether r' is a serial extension of r_t^i or a serial extension of r_t^{i+1}. Consequently, if r_t^i is 0-valent then so is r_t^{i+1}: a contradiction.

Therefore, $p'_{i+1} \neq p'_1$. Process p'_{i+1} is the only process which can distinguish r_t^i from r_t^{i+1} at the end of round t: p'_{i+1} receives a t-round message from p'_1 in r_t^i and does not receive a t-round message from p'_1 in r_t^{i+1}. *For convenience of presentation let us denote p'_{i+1} by p_x and p'_1 as p_y.*

We now construct two synchronous runs s^1 and s^0 in which p_x decides different values.

- s^1: This run is a one round serial extension of r_t^{i+1} in which no process crashes at round $t+1$. Since partial run r_t^{i+1} is 1-valent and s^1 is a serial $(t+1)$-round partial run, p_x decides 1 at the end of round $t+1$.
- s^0: This run is a one round serial extension of r_t^i in which no process crashes at round $t+1$. Since partial run r_t^i is 0-valent and s^0 is a serial $(t+1)$-round partial run, process p_x decides 0 at the end of round $t+1$.

We now construct two $(t+1)$-round asynchronous partial runs a^0 and a^1 (these runs correspond to the asynchronous partial runs r and r' mentioned in Section 3.1).

- a^1: This is an asynchronous $(t+1)$-round partial run which is defined as follows for each round k:

  - $k \le t-1$: The partial run is identical to the first $t-1$ rounds of s^1.
  - $k = t$: No process crashes in this round. Unlike s^1, p_y does not crash in round t of this partial run. But, every process (except p_y) receives the same set of messages as in round t of s^1. (Any process which does not receive a message from p_y in round t of this run, falsely suspects p_y.) Process p_y receives some arbitary set of messages, $M_{a^1}(y,t)$.

    Observations: (1) At the end of round t of a^1, only p_y can distinguish the first t rounds of a^1 from the first t rounds of s^1. (2) At most $t-1$ processes has crashed in first t round of a^1. To see why, notice that the first $t-1$ rounds of a^1 is identical to first $t-1$ rounds of s^1. As s^1 is a serial run, at most $t-1$ processes can crash in first $t-1$ rounds of s^1 (and a^1). No process crashes in round t of a^1.
  - $k = t+1$: Due to false suspicion, (1) processes distinct from p_x, do not receive any message from p_x, and (2) p_x does not receive any message from p_y. Process p_x cannot distinguish this partial run from s^1, and therefore, decides 1 at the end of this round and then crashes.

    Observations: (1) No process suspects more than t processes in a round: in round t and $t+1$ every process suspects at most $t-1$ processes which have already crashed in first $t-1$ rounds, and either p_x or p_y. (2) To see why p_x cannot distinguish between a^1 and s^1, recall that no process (except p_y) can distinguish first t rounds of a^1 from that of s^1. Therefore, every process (except p_y) sends the same message in round $t+1$ of both partial runs. As p_x does not receive any message from p_y in round $t+1$ of both runs, it receives identical messages in round $t+1$ of both runs.

- a^0: This is an asynchronous $(t+1)$-round partial run which is defined as follows for each round k:

  - $k \le t-1$: The partial run is identical to the first $t-1$ rounds of s^0.
  - $k = t$: No process crashes in this round. Unlike s^0, p_y does not crash in round t of this partial run. But, every process (except p_y) receives the same set of messages as in round t of s^0. (Any process which does not receive a message from p_y in round t of this run, falsely suspects p_y.) Process p_y receives the same set of messages as in a^1, $M_{a^1}(y,t)$.
  - $k = t+1$: Due to false suspicion, (1) processes distinct from p_x, do not receive any message from p_x, and (2) p_x does not receive any message from p_y. Process p_x cannot distinguish this partial run from s^0, and therefore, decides 0 at the end of this round and then crashes.

    Observations: It is easy to verify that (1) no process suspects more than t processes in a round, and (2) p_x cannot distinguish between a^0 and s^0. Furthermore, no process which is up at the end of the two partial runs (a^1 and a^0) can distinguish the two runs. To see why, notice that, at the end of round t, only p_x can distinguish between the partial runs: p_x receives $m(y,t)$ in a^0 and does not receive $m(y,t)$ in a^1. In round $t+1$ of both runs, processes (other that p_x) does not receive any message from p_x. Thus, p_x is the only process which can distinguish a^1 from a^0, and it crashes at the end of both partial runs.

Thus, we have constructed two $(t+1)$-round partial runs a^0 and a^1, which are indistinguishable to all processes which are up at the end of round $t+1$, and there is a process which decides different values and then crashes in a^0 and a^1. Consider a run $r_{0,1}$ which extends a^1. By consensus agreement, every correct process eventually decides 1 in this run. Now we replace first $t+1$ rounds of $r_{0,1}$ by a^0. As no process which is up after round $t+1$ can distinguish a^0 from a^1, so correct processes still decide 1 in modified $r_{0,1}$: violating consensus agreement, as p_x decides 0 in a^0. □

B Correctness of the Consensus Algorithm of Figure 3

Validity and termination properties of A_{t+2} are straightforward. We focus here on the elimination property (from which uniform agreement, integrity, and fast decision properties can be derived easily). For convenience of discussion, we introduce the following notation. Given any variable x_i at process p_i, we denote by $x_i[k_i]$ the value of the variable x_i immediately after the completion of procedure compute() at round k_i ($1 \leq k_i \leq t+2$). If p_i does not invoke procedure compute(), or fails to return from the procedure at round k_i (maybe because p_i has crashed in a lower round), then $x_i[k_i]$ is *undefined*. For example, $est_i[1]$ is the value of est_i just after line 5 at round 1 and $est_i[t+2]$ is the value of est_i just after line 10 at round $t+2$.

Lemma 6. *(Elimination) If there are two distinct processes p_i and p_j such that (1) p_i and p_j send* NEWESTIMATE *messages, (2) $nE_i[t+2] \neq \perp$, and (3) $nE_j[t+2] \neq \perp$, then $nE_i[t+2] = nE_j[t+2]$.*

Proof: Suppose by contradiction that there exist two distinct processes p_i and p_j such that, (1) $nE_i[t+2] = c \neq \perp$, (2) $nE_j[t+2] = d \neq \perp$, and (3) $c \neq d$. We prove four lemmata (Lemma 7 to Lemma 10) based on this assumption. Lemma 10 contradicts Lemma 8.

Without loss of generality we can assume that $c < d$. For a run of A_{t+2} we define set C_k as follows. C_1 is the set of processes whose proposal values are less than or equal to c and C_k ($2 \leq k \leq t+2$) is the set $C_1 \cup \{p_j \mid \exists k' \leq k, est_j[k'] \leq c\}$. From the definition of C_k, we can immediately make the following three observations for the given run of A_{t+2}.

O1: $|C_1| \geq 1$. Otherwise, if every process proposes a value greater than c, then $nE_i[t+2]$ must be different from c.

O2: For $1 \leq k \leq t+1$, $C_k \subseteq C_{k+1}$. This follows directly from the definition of C_k.
O3: For $1 \leq k \leq t+1$, $\forall p_i \in C_k$, if p_i sends an ESTIMATE message in any round $k' \geq k$ then $est_i[k'] \leq c$. A process always receives its own ESTIMATE message, so the updated est in line 27 is always less than or equal to previous est.

Lemma 7: *Consider the state of any process p_l after completing procedure compute() at round k ($2 \leq k \leq t+2$). Let $senderMS_l[k]$ be the set of processes which are the sender of the messages in $msgSet_l[k]$. Then, $senderMS_l[k] = \Pi - Halt_l[k]$.*
Proof: Consider any process $p_m \in \Pi$. There are three exhaustive and mutually exclusive cases regarding messages from p_m to p_l in round $k-1$ ($2 \leq k \leq t+2$):
- If p_l does not receive an ESTIMATE message from p_m at round $k-1$, then $p_m \in Halt_l[k]$ (line 24) and $p_m \notin senderMS_l[k]$.
- If p_l receives an ESTIMATE message from p_m and $p_m \notin Halt_l[k-1]$, then $p_m \in senderMS_l[k]$ (line 23) and $p_m \notin Halt_l[k]$.
- If p_l receives an ESTIMATE message from p_m and $p_m \in Halt_l[k-1]$, then $p_m \notin senderMS_l[k]$ and $p_m \in Halt_l[k]$ (line 24). Thus, a process is either in $senderMS_l[k]$ or $Halt_l[k]$. $\Box$

Lemma 8: $|C_{t+1}| \leq t$.
Proof: Suppose by contradiction that $|C_{t+1}| > t$. Consider any process $p_m \in C_{t+1}$. From Observation O3, it follows that either p_m sends an ESTIMATE message with $est \leq c$ at round $t+1$ or does not send any ESTIMATE message (if p_m crashes). Now consider the messages received by process p_j at round $t+1$. (Recall that $nE_j[t+2] = d > c$.) The set $msgSet_j[t+2]$ does not contain any message from p_m. Otherwise, $nE_j[t+2]$ must be less that or equal to c. Therefore, from Lemma 7 it follows that $p_m \in Halt_j[t+2]$. Consequently, $C_{t+1} \subseteq Halt_j[t+2]$, and $|Halt_j[t+2]| \geq |C_{t+1}| > t$. It thus follows from line 29 that $nE_j[t+2]$ is $\perp$: a contradiction. $\Box$

Lemma 9: *$p_i \in C_{t+2}$ and $p_i \notin C_t$.*
Proof: Notice that $nE_i[t+2] = c \neq \perp$ implies that $est_i[t+2] = c$ (line 32). Thus, from the definition of C_{t+2} it follows that $p_i \in C_{t+2}$.

For the next part of the proof, suppose by contradiction that $p_i \in C_t$. Consider any process $p_m \in \Pi - C_{t+1}$. From the definition of C_{t+1}, we know that $est_m[t+1] > c$. Therefore, $msgSet_m[t+1]$ does not contain any estimate message from p_i. (Otherwise, on receving $est \leq c$ from p_i, p_m has to set $est_m[t+1] \leq c$.) Therefore, from Lemma 7 it follows that $p_i \in Halt_m[t+1]$. Furthermore, every process in $\Pi - C_{t+1}$ either crashes or sends an (ESTIMATE, $t+1$, $*$, $Halt'$) message such that $p_i \in Halt'$.

As $nE_i[t+2] \neq \perp$, so we know that $mistake_i[t+2] = false$. This implies that p_i has not received any (ESTIMATE, $t+1$, $*$, $Halt'$) message such that $p_i \in Halt'$. Therefore, $\Pi - C_{t+1} \subseteq Halt_i[t+2]$. From Lemma 8 it follows that $|\Pi - C_{t+1}| \geq n - t > t$ (recall that $t < \lceil \frac{n}{2} \rceil$). So, $|Halt_i[t+2]| > t$: a contradiction with $nE_i[t+2] \neq \perp$. $\Box$

```
07.a. if k = 2 then
07.b.   if every received (ESTIMATE, 2, est, Halt) message
        has Halt = ∅ then
07.c.     vc_i ← any est value received
07.d.     if received (ESTIMATE, 2, est, Halt) message
          from all processes in Π then
07.e.       decide(vc_i)
```

Figure 5: Optimizer for A_{t+2}

Lemma 10: *(1) For all k such that* $1 \le k \le t$*:* $C_k \subset C_{k+1}$*. (*C_k *is a proper subset of* C_{k+1}*). (2)* $|C_{t+1}| \ge t+1$*.*

Proof: (1) Consider any $1 \le k \le t$. Recall from Observation O2, $C_k \subseteq C_{k+1}$. Thus, either $C_k \subset C_{k+1}$ or $C_k = C_{k+1}$. Suppose by contradiction that $C_k = C_{k+1}$.

For any process $p_m \in \Pi - C_{k+1}$, $msgSet_m[k+1]$ does not contain an (ESTIMATE, k, $*$, $*$) message from any process in C_k; otherwise, $est_m[k+1]$ must be less than or equal to c and $p_m \in C_{k+1}$. Therefore, from Lemma 7, $C_k \subseteq Halt_m[k+1]$. Since $C_k = C_{k+1}$, so $C_{k+1} \subseteq Halt_m[k+1]$. Thus, in subsequent rounds, processes in $\Pi - C_{k+1}$ ignore all messages from any process in C_{k+1} while updating *est*, and therefore *est* is always greater than c (at processes in $\Pi - C_{k+1}$). Therefore, after round $k+1$, the set C never changes (no process in $\Pi - C$ ever adopts a value less than or equal to c as its *est*), i.e., $C_{k+1} = C_{k+2} = ... = C_{t+2}$. A contradiction with Lemma 9.

(2) Part (1) of this lemma implies that $1 \le k \le t, |C_{k+1}| - |C_k| \ge 1$. We know from Observation O1 that $|C_1| \ge 1$. Therefore, $|C_{t+1}| \ge t+1$. □

C An optimization

Algorithm A_{t+2} can be improved to achieve a global decision at round 2 in every failure-free synchronous run (commonly known as *nice runs*). At the end of round 2, if any process p_i is certain that there were no suspicions in round 1 (p_i receives round 2 messages from each of the n processes with $Halt = \emptyset$) then p_i decides immediately on any *est* value received and sets the proposal variable vc_i for the underlying consensus algorithm C to that value. Otherwise, if p_i does not detect any suspicion at round 1 (p_i does not receive round 2 messages from all n processes, however, every round 2 messages received by p_i has $Halt = \emptyset$) then p_i sets vc_i to any *est* value received. Figure 5 describes the modification more precisely. For the optimization, the lines in Figure 5 are inserted between line 7 and line 8 of Figure 3.

It is straightforward to see that Figure 5 performs the required optimization without violating any of the consensus properties or the fast decision property. Suppose that some process p_i decides d at round 2. To see why consensus agreement is not violated, notice that p_i decides in line 7.e only if there has been a complete exchange of estimate messages at round 1 (i.e., no process suspected any process). As the proposal values form a totally ordered set, every ESTIMATE message at round 2 had the same *est* value d (d is precisely the minimum of all proposed values), and therefore, every message sent at round 2 is (ESTIMATE, 2, d, $\emptyset$). Thus, the only possible decision value at round 2 is d, and processes set both vc_i and est_i to d. Therefore, any process which decides at round $t + 2$, decides d and any process which invokes $\text{propose}_C(*)$, does so with value d. Agreement is obvious.