Randomization and Failure Detection: A Hybrid Approach to Solve Consensus*

Marcos K. Aguilera and Sam Toueg

aguilera@cs.cornell.edu sam@cs.cornell.edu

Department of Computer Science
Upson Hall, Cornell University
Ithaca, NY 14853-7501, USA.

June 27, 1996

Abstract

We present a Consensus algorithm that combines randomization and unreliable failure detection, two well-known techniques for solving Consensus in asynchronous systems with crash failures. This hybrid algorithm combines advantages from both approaches: it guarantees deterministic termination if the failure detector is accurate, and probabilistic termination otherwise. In executions with no failures or failure detector mistakes, the most likely ones in practice, Consensus is reached in only two asynchronous rounds.

1 Background

It is well-known that Consensus cannot be solved in asynchronous systems with failures, even if communication is reliable, at most one process may fail, and it can only fail by crashing. This "impossibility of Consensus", shown in a seminal paper by Fischer, Lynch and Paterson [FLP85], has been the subject of intense research seeking to "circumvent" this negative result (e.g., [Ben83, BT83, Rab83, DDS87, DLS88, CT96, CHT96]).

We focus on two of the major techniques to circumvent the impossibility of Consensus in asynchronous systems: randomization and unreliable failure detection. The first one assumes that each process has an oracle (denoted *R-oracle*) that provides *random bits* [Ben83]. The second technique assumes that each process has an oracle (denoted *FD-oracle*) that provides *a list of processes suspected to have crashed* [CT96]. Each approach has some advantages over the other, and we seek to combine advantages from both.

*Research partially supported by NSF grant CCR-9402896 and DARPA/NASA Ames grant NAG-2-593

With a randomized Consensus algorithm, every process can query its R-oracle, and use the oracle's random bit to determine its next step. With such an algorithm, termination is achieved with probability 1, within a finite expected number of steps (for a survey of randomized Consensus algorithms see [CD89]).

With a failure-detector based Consensus algorithm, every process can query its local FD-oracle (which provides a list of processes that are suspected to have crashed) to determine the process's next step. Consensus can be solved with FD-oracles that make an infinite number of mistakes. In particular, Consensus can be solved with any FD-oracle that satisfies two properties, *strong completeness* and *eventual weak accuracy*. Roughly speaking, the first property states that every process that crashes is eventually suspected by every correct process, and the second one states that some correct process is eventually not suspected. These properties define the weakest class of failure detectors that can be used to solve Consensus [CHT96].

In this paper we describe a hybrid Consensus algorithm with the following properties. Every process has access to both an R-oracle and an FD-oracle. If the FD-oracle satisfies the above two properties, the algorithm solves Consensus (no matter how the R-oracle behaves). If the FD-oracle loses its accuracy property, but the R-oracle works, the algorithm still solves Consensus, albeit "only" with probability 1. In executions with no failures or failure detector mistakes, the most likely ones in practice, the algorithm reaches Consensus in two asynchronous rounds. A discussion of the relative merits of randomization, failure detection, and this hybrid approach is postponed to Section 7.

The idea of combining randomization and failure detection to solve Consensus in asynchronous systems first appeared in [DM94]. A related idea, namely, combining randomization and deterministic algorithms to solve Consensus in synchronous systems was explored in [GP90, Zam96]. A brief comparison with our results is given in Section 8.

2 Informal Model

Our model of asynchronous computation is patterned after the one in [FLP85], and its extension in [CHT96]. We only sketch its main features here. We consider *asynchronous* distributed systems in which there is no bound on message delay, clock drift, or the time necessary to execute a step. To simplify the presentation of our model, we assume the existence of a discrete global clock. This is merely a fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers $\mathbb{N}$.

The system consists of a set of n *processes*, $\Pi = \{p_0, p_1, \ldots, p_{n-1}\}$. Every pair of processes is connected by a reliable communication channel. Up to f processes can fail by *crashing*. A failure pattern indicates which processes crash, and when, during an execution. Formally, a *failure pattern* F is a function from $\mathbb{N}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed through time t. Once a process crashes, it does not "recover", i.e., $\forall t : F(t) \subseteq F(t+1)$. We define $\text{crashed}(F) = \bigcup_{t \in \mathbb{N}} F(t)$ and $\text{correct}(F) = \Pi - \text{crashed}(F)$. If $p \in \text{crashed}(F)$ we say *p crashes (in F)* and if $p \in \text{correct}(F)$ we say *p is correct (in F)*.

Each process has access to two oracles: a failure detector, henceforth denoted the *FD-oracle*, and a random number generator, henceforth denoted the *R-oracle*. When a process queries its FD-oracle, it obtains a list of processes. When it queries its R-oracle it obtains a bit. The properties of these oracles are described in the two next sections.

A distributed algorithm $\mathcal{A}$ is a collection of n deterministic automata (one for each process in the system) that communicate by sending messages through reliable channels. The execution of $\mathcal{A}$ occurs in *steps* as follows. For every time $t \in \mathcal{T}$, at most one process takes a step. Each step consists of receiving a message; querying the FD-oracle; querying the R-oracle; changing state; and optionally sending a message to one process. We assume that messages are never lost. That is, if a process does not crash, it eventually receives every message sent to it.

A schedule is a sequence $\{s_j\}_{j\in\mathbb{N}}$ of processes and a sequence $\{t_j\}_{j\in\mathbb{N}}$ of strictly increasing times. A schedule indicates which processes take a step and when: for each j, process s_j takes a step at time t_j. A schedule is *consistent (with respect to a failure pattern F)* if a process does not take a step after it has crashed (in F). A schedule is *fair (with respect to a failure pattern F)* if each process that is correct (in F) takes an infinite number of steps. We consider only schedules that are consistent and fair.

2.1 FD-oracles

Every process p has access to a local FD-oracle module that outputs a list of processes that are suspected to have crashed. If some process q belongs to such list, we say that *p suspects q*.[1] FD-oracles can make mistakes: it is possible for a process p to be suspected by another even though p did not crash, or for a process to crash and never be suspected. FD-oracles can be classified according to properties that limit the extent of such mistakes. We focus on one of the eight classes of FD-oracles defined in [CT96], namely, the class of *Eventually Strong* failure detectors, denoted $\diamond S$. An FD-oracle belongs to $\diamond S$ if and only if it satisfies two properties:

Strong completeness: Eventually every process that crashes is permanently suspected by *every* correct process (formally, $\exists t \in \mathcal{T}, \forall p \in \text{crashed}(F), \forall q \in \text{correct}(F), \forall t' \geq t : p \in \text{FD}_q^{t'}$, where $\text{FD}_q^{t'}$ denotes the output of q's FD-oracle module at time t').

Eventual weak accuracy: There is a time after which some correct process is never suspected by any correct process (formally, $\exists t \in \mathcal{T}, \exists p \in \text{correct}(F), \forall t' \geq t, \forall q \in \text{correct}(F) : p \notin \text{FD}_q^{t'}$).

It is known that $\diamond S$ is the weakest class of FD-oracles that can be used to solve Consensus.

2.2 R-oracles

Each process has access to a local R-oracle module that outputs one bit each time it is queried. We say that the R-oracle is *random* if it outputs an independent random bit for each query. For simplicity, we assume a uniform distribution, i.e., a random R-oracle outputs 0 and 1, each with probability 1/2.

[1]In general, processes do not have to agree on the list of suspects at any one time or ever.

2.3 Adversary Power

When designing fault-tolerant algorithms, we often assume that an intelligent adversary has some control on the behavior of the system, e.g., the adversary may be able to control the occurrence and the timing of process failures, the message delays, and the scheduling of processes. Adversaries may have limitations on their computing power and on the information that they can obtain from the system. Different algorithms are designed to defeat different types of adversaries [CD89].

We now describe the adversary that our hybrid algorithm defeats. The adversary has unbounded computational power, and full knowledge of all process steps that already occurred. In particular, it knows the contents of all past messages, the internal state of all processes in the system,[2] and all the previous outputs of both the R-oracle and FD-oracle. With this information, at any time in the execution, the adversary can dynamically select which process takes the next step, which message this process receives (if any), and which processes (if any) crash. The adversary, however, operates under the following restrictions: the final schedule must be consistent and fair, every message sent to a correct process must be eventually received, and at most f processes may crash over the entire execution.

In addition to the above power, we allow the adversary to initially select *one* of the two oracles to control, and possibly corrupt.[3] If the adversary selects to control the R-oracle, it can predict and even determine the bits output by that oracle. For example, the adversary can force some local R-oracle module to always output 0, or it can dynamically adjust the R-oracle's output according to what the processes have done so far.

If the adversary selects to control the FD-oracle, it can ensure that the FD-oracle does not satisfy eventual weak accuracy. In other words, at *any* time the adversary can include *any* process (whether correct or not) in the output of the local FD-oracle module of any process. The adversary, however, does not have the power to disrupt the strong completeness property of the FD-oracle. This is not a limitation in practice: most failure detectors are based on time-outs and eventually detect all process crashes.

If the adversary does not control the R-oracle then the R-oracle is random. If the adversary does not control the FD-oracle then the FD-oracle is in $\diamond S$. We stress that the algorithm does *not* know which one of the two oracles (FD-oracle or R-oracle) is controlled by the adversary.

3 The Consensus Problem

Uniform Binary Consensus is defined in terms of two primitives, propose(v) and decide(v), where $v \in \{0, 1\}$. When a process executes propose(v), we say that it *proposes* v; similarly, when a process executes decide(v), we say that it *decides* v. The Uniform Binary Consensus problem is specified as follows:

[2]This is in contrast to the assumptions made by several algorithms, e.g., those that use cryptographic techniques.

[3]From the definitions of these oracles, it is clear that we can allow the adversary to control the behavior of both oracles for an arbitrary but finite amount of time. The only restriction is that it must eventually stop controlling one of the oracles.

Uniform agreement: If processes p and p' decide v and v', respectively, then $v = v'$;

Uniform validity: If a process decides v, then v was proposed by some process;

Termination: Every correct process eventually decides some value.

For probabilistic Consensus algorithms, Termination is weakened to

Termination with probability 1: With probability 1, every correct process eventually decides some value.

4 Hybrid Consensus Algorithm

The hybrid Consensus algorithm shown in Figure 1 combines Ben-Or's algorithm [Ben83] with failure-detection and the rotating coordinator paradigm used in [CT96]. With this paradigm, we assume that all processes have *a priori* knowledge that during phase k, one selected process, namely $p_{k \bmod n}$, is the coordinator. The algorithm works under the assumption that a majority of processes are correct (i.e., $n > 2f$). It is easy to see that this requirement is necessary for any algorithm that solves Consensus in asynchronous systems with crash failures, even if all processes have access to a random R-oracle and an FD-oracle that belongs to $\Diamond S$.

In the hybrid algorithm, every message contains a tag (R, P, S or E), a phase number, and a value which is either 0 or 1 (for messages tagged P or S, it could also be "?"). Messages tagged R are called *reports*; those tagged with P are called *proposals*; those with tag S are called *suggestions [to the coordinator]*; those with tag E are called *estimates [from the coordinator]*. When p sends (R, k, v), (P, k, v) or (S, k, v) we say that p *reports*, *proposes* or *suggests* v in phase k, respectively. When the coordinator sends (E, k, v) we say that the coordinator sends estimate v in phase k.

Each execution of the **while** loop is called a *phase*, and each phase consists of four asynchronous rounds. In the first round, processes report to each other their current estimate (0 or 1) for a decision value.

In the second round, if a process receives a majority of reports for the *same* value then it proposes that value to all processes, otherwise it proposes "?". Note that it is impossible for one process to propose 0 and another process to propose 1. At the end of the second round, if a process receives $f + 1$ proposals for the same value different than ?, then it decides that value. If it receives at least one value different than ?, then it adopts that value as its new estimate, otherwise it adopts ? for estimate.

In the third round, processes suggest their estimate to the current coordinator. If the coordinator receives a value different than ? then it sends that value as its estimate. Otherwise, the coordinator queries the R-oracle, and sends the random value that it obtains as its estimate.

In the fourth round, processes wait until they receive the coordinator's estimate or until their FD-oracle suspects the coordinator. If a process receives the coordinator's estimate, it

Every process p executes the following:

```
procedure propose(v_p)                          {v_p is the value proposed by process p}
    x ← v_p                                     {x is p's current estimate of the decision value}
    k ← 0
    while true do
        k ← k + 1                               {k is the current phase number}
        c ← p_{k mod n}                         {c is the current coordinator}
        send (R, k, x) to all processes
        wait for messages of the form (R, k, *) from n − f processes {"*" can be 0 or 1}
        if received more than n/2 (R, k, v) with the same v
        then send (P, k, v) to all processes
        else send (P, k, ?) to all processes
        wait for messages of the form (P, k, *) from n − f processes {"*" can be 0, 1 or ?}
        if received at least f + 1 (P, k, v) with the same v ≠ ? then decide(v)
        if at least one (P, k, v) with v ≠ ? then x ← v else x ← ?
        send (S, k, x) to c
        if p = c then
            wait for messages of the form (S, k, *) from n − f processes
            if received at least one (S, k, v) with v ≠ ?
            then send (E, k, v) to all processes
            else
                random_bit ← R-oracle                   {query R-oracle}
                send (E, k, random_bit) to all processes
        wait until receive (E, k, v_coord) from c or c ∈ FD-oracle   {query FD-oracle}
        if received (E, k, v_coord)
        then x ← v_coord
        else if x = ? then x ← R-oracle                 {query R-oracle}
```

Figure 1: Hybrid Consensus algorithm

adopts it. Otherwise, if its current estimate is ?, it adopts a random value obtained from its R-oracle.

To simplify the presentation, the algorithm in Figure 1 does not include a halt statement. Moreover, once a correct process decides a value, it will keep deciding the same value in all subsequent phases. However, it is easy to modify the algorithm so that every process decides at most once, and halts at most one round after deciding.

5 Proof of Correctness

The hybrid algorithm shown in Figure 1 always satisfies the safety properties of Consensus. This holds no matter how the FD-oracle or the R-oracle behave, that is, even if these oracles are totally under the control of the adversary. On the other hand, the algorithm satisfies liveness properties only if the FD-oracle satisfies strong completeness. Strong completeness is easy to achieve in practice: most failure-detectors use time-out mechanisms, and every process that crashes eventually causes a time-out, and therefore a permanent suspicion.

Assume that there is a majority of correct processes (i.e., $n > 2f$). We show the following:

Theorem 1

(Safety) The hybrid algorithm always satisfies validity and uniform agreement.

(Liveness) Suppose that the FD-oracle *satisfies strong completeness.*

- *If the* FD-oracle *satisfies eventual weak accuracy, i.e., it is in $\diamond S$, then the algorithm satisfies termination.*
- *If the* R-oracle *is random then the algorithm satisfies termination with probability 1.*

Proof: We say that *process p starts phase k* if process p completes at least $k-1$ iterations of the **while** loop. We say that *process p reaches line n in phase k* if process p starts phase k and p executes past line $n-1$ in that phase. We say that *v is k-locked* if every process that starts phase k does so with its variable x set to v. When ambiguities may arise, a local variable of a process p is subscripted by p, e.g., x_p is the local variable x of process p.

We first show the safety properties.

Lemma 1 *Suppose $k > 0$. Then (1) it is impossible for a process to propose* 0 *and another one to propose* 1 *in the same phase k; and (2) it is impossible for a process to suggest* 0 *and another to suggest* 1 *in the same phase k.*

Proof: We prove (1) by contradiction: suppose that two processes p and q propose 0 and 1, respectively, in phase k. Thus, p received more than $n/2$ reports for 0 and q received more than $n/2$ reports for 1 in phase k. But then there is a process that reports 0 to p and 1 to q in phase k, and this is impossible. This proves (1).

Now (2) follows from (1) since if a process suggests $v \neq$? in phase k, then v was proposed in phase k. □

Lemma 2 *If some process decides v in phase $k > 0$, then v is $(k+1)$-locked.*

Proof: Suppose some process p decides v in phase $k > 0$ (note that $v \neq ?$), and let q be any process that starts phase $k+1$. From the algorithm, p receives at least $f+1$ proposals for v in phase k (line 12). Let r be any process that suggests a value in line 14 of phase k. Before suggesting (line 14), r waits for $n-f$ proposals in line 11. Because p receives $f+1$ proposals for v, r must have received at least one proposal for v. Moreover, by Lemma 1, r does not receive any proposals for $\bar{v}$. [4] So r sets x_r to v in line 13 and suggests v in phase k. Thus, (1) q sets x_q to v in line 13, and (2) the coordinator of phase k can only receive suggestions for v. In particular, the coordinator does not receive ?. So, if the coordinator sends an estimate in phase k (line 18), that estimate is also v. If q receives that estimate (line 22), then q resets x_q to v in line 24. Otherwise q does not modify x_q (because x_q is different than ?). In either case, q starts phase $k+1$ with $x_q = v$. □

Lemma 3 *If a value v is k-locked for some $k > 0$, then every process that reaches line 13 in phase k decides v in phase k.*

Proof: Suppose v is k-locked for some $k > 0$. Then, all reports sent in line 6 of phase k are for v. Since $n - f > n/2$, every process that proposes some value in phase k proposes v in line 9. Consider a process p that reaches line 13 in phase k. Clearly, p receives $n-f$ proposals (line 11) for v in phase k. Since $n - f \geq f + 1$, p decides v in phase k. □

Corollary 1 *If some process decides v in phase $k > 0$, then every process that reaches line 13 in phase $k+1$ decides v in phase $k+1$.*

Proof: By Lemma 2 and Lemma 3. □

Corollary 2 *(Uniform agreement) If some processes p and p' decide v and v' in phase $k > 0$ and $k' > 0$, respectively, then $v = v'$.*

Proof: For $k = k'$ the result follows from Lemma 1 and the fact that a process can decide a value in a phase only if that value was proposed in the same phase. Assume that $k < k'$. Since p' decides in phase k' then p' reaches line 13 in every phase r, $k < r \leq k'$. Since p decides v in phase k, by Corollary 1 p' decides v in phase $k+1 \leq k'$. By additional applications of Corollary 1, we conclude that p' decides v in phase k'. Each process can decide at most once per phase, so $v = v'$. □

Corollary 3 *(Validity) If some process p decides some value v then v was the initial value of some process.*

Proof: Note $v \in \{0, 1\}$. If the initial values of all processes are not identical, then v is clearly the initial value of some process. Now, suppose all processes have the same initial value w. Thus, w is 1-locked. From Lemma 3, p decides w, and from Corollary 2, $w = v$. □

From now on we assume that the FD-oracle satisfies strong completeness, and proceed to prove the liveness properties.

[4] We denote by $\bar{v}$ the binary complement of bit v.

Lemma 4 *Every correct process starts every phase $k > 0$.*

Proof: The detailed proof is by a simple but tedious induction on k. We describe only the central idea here. In each phase, there are four **wait** statements that can potentially block processes (lines 7, 11, 16, 22). It is not possible for a correct process to be blocked forever in any of the first three **wait** statements, because at least $n - f$ processes are correct and send the messages that this process is waiting for. Consider the fourth **wait** statement. Either the coordinator c sends its estimate to all processes or c crashes. In the first case, every correct process receives this estimate. In the second case, c eventually appears on the list of suspects, i.e., $c \in$ *FD-oracle* (because the *FD-oracle* satisfies strong completeness). So no correct process waits forever at the fourth **wait** statement of a phase. □

Corollary 4 *If a value v is k-locked for some $k > 0$, then every correct process decides v in phase k.*

Proof: Immediate from Lemmata 3 and 4. □

Corollary 5 *If some process decides v in phase $k > 0$, then every correct process decides v in phase $k + 1$ (and thus in all subsequent phases).*

Proof: Immediate from Corollary 1 and Lemma 4. □

Lemma 5 *(Termination) If the* FD-oracle *satisfies eventual weak accuracy then every correct process decides.*

Proof: By eventual weak accuracy of the FD-oracle, there exists a time after which a correct process p_m is never suspected by anyone. For each process p_i, let k_i be the value of variable k of process p_i at that time. Let $\bar{k} = 1 + max_i\{k_i\}$ and $\hat{k} = min\{j + \bar{k} : j \in \mathbb{N} \wedge (j + \bar{k}) \bmod n = m\}$. Let q and r be arbitrary processes that start phase $\hat{k} + 1$. In phase $\hat{k}$, the coordinator is process p_m, by choice of $\hat{k}$. So q and r do not suspect the coordinator and thus q and r set x_q and x_r to the coordinator's estimate in line 24. Since this estimate is different from ? and unique for phase $\hat{k}$, we have $x_q = x_r = v$ for some $v \neq$? at the beginning of phase $\hat{k} + 1$. So v is $\hat{k} + 1$-locked. Therefore, by Corollary 4, all correct processes decide v in phase $\hat{k} + 1$. □

We now proceed to show that if the R-oracle is random, then the algorithm satisfies termination with probability 1. For $k > 0$, let τ_k be the first time that any process receives $n - f$ proposals in phase k. From Lemma 4, for every $k > 0$, some process receives $n - f$ proposals in phase k, and so τ_k is well-defined. Note that in our algorithm no process queries the R-oracle in phase k before time τ_k.

For each $k > 0$, we say that a value *v is k-major at time t* if by time t more than $n/2$ processes have started phase k with their variable x set to v. Clearly, for each $k > 0$ and all times t and t', it is impossible for 0 to be k-major at t, and 1 to be k-major at t'.

We say that *a process p R-gets v in phase k at time t* if either:

- In phase k at time t, p obtains v from the R-oracle in line 25 and sets x_p to v; or
- In phase k, the coordinator obtains v from the R-oracle in line 20, sends v as its estimate to all processes, p receives this estimate and sets x_p to v in line 24 at time t.

---

function FavorableToss(r, u): **bit** $\{$evaluated only at time $u \geq \tau_k$ where $k = 2r\}$

$k \leftarrow 2r$ $\{k$ is the first phase in epoch $r\}$

if some value v is k-major at time τ_k **then return** v

if by time u no process received $n - f$ proposals in phase $k+1$ **then return** 0 $\{u < \tau_{k+1}\}$

if before time τ_{k+1}: $\{$here $u \geq \tau_{k+1}\}$
 (a) 1 is k-major, and
 (b) less than $n/2$ processes R-got a value in phase k, and
 (c) the coordinator did not query the R-oracle in line 20 of phase k
then return 1
else return 0

Figure 2: Favorable coin toss algorithm

Intuitively, a process p R-gets v if p sets x_p to v, and p obtained v from an R-oracle query (directly, or indirectly through the coordinator).

Lemma 6 *If at time t a process p starts phase $k > 1$ with x_p set to some value v, then either v is $(k-1)$-major at time t or p R-gets v in phase $k-1$.*

Proof: Consider phase $k-1$. Suppose p did not R-get v. Let t' be the last time p updates x_p in phase $k-1$. Note that $t' < t$. Then, at time t', either (a) p receives the estimate from the coordinator, and the coordinator obtained that estimate from one of its non-? suggestions; or (b) p sets x_p in line 13. In both cases, more than $n/2$ processes must have reported v in phase $k-1$ before time t'. Therefore, at least $n/2$ processes have started phase $k-1$ by time t' (and thus by time t) with their variable x set to v. $\Box$

For the rest of the proof, we group pairs of phases into *epochs* as follows: *epoch r* consists of phases $2r$ and $2r+1$.[5] We will define the concept of a "lucky" epoch — one in which processes toss coins that cause the termination of the algorithm (no matter what the adversary does). To do so, we first define function *FavorableToss*(r, u) given in Figure 2. We say that *epoch r is lucky* if, for every process p and any time u, if p queries the R-oracle in epoch r at time u, then p obtains *FavorableToss*(r, u) from the R-oracle. Note that if p queries the R-oracle in epoch r at time u, this occurs after at least one process receives $n-f$ proposals in phase $2r$. Thus, $\tau_{2r} \leq u$, so the value of *FavorableToss*(r, u) depends only on what occurred in the system up to time u.

Lemma 7 *If the R-oracle is random then the probability that some epoch is lucky is 1.*

Proof: The result is immediate from the following observation: for every $r \geq 1$, (a) the probability that epoch r is lucky is at least $2^{-(2n+2)}$ (because in each phase there are at most

[5] Phase 1 is not part of any epoch.

$n+1$ queries to the R-oracle, and the R-oracle is random), and (b) for any $r' \neq r$, the events "epoch r is lucky" and "epoch r' is lucky" are independent (because epochs r and r' consist of disjoint sets of phases).□

Lemma 8 *For every $r \geq 1$, if epoch r is lucky then some value is $(2r+1)$-locked or $(2r+2)$-locked.*

Proof: Throughout the proof of this lemma, fix some arbitrary $r \geq 1$ and assume that epoch r is lucky. Let $k = 2r$; recall that epoch r consists of phases k and $k+1$. Since epoch r is lucky, if any process R-gets a value v at time t and in phase $j = k$ or $j = k+1$, then $v = \mathit{FavorableToss}(r, u)$ for some time u, $\tau_j \leq u \leq t$ (value v was obtained either directly from the R-oracle or indirectly through the coordinator).

Case 1: Suppose some value v is k-major at time τ_k. By the definition of *FavorableToss*, for any u such that $\tau_k \leq u$, $\mathit{FavorableToss}(r, u) = v$. So, if a process R-gets a value in phase k, that value is v. Note that $\bar{v}$ is not k-major at any time. By Lemma 6, v is $(k+1)$-locked.

Case 2: Now assume that no value is k-major at time τ_k.

Case 2.1: Suppose that no value is k-major before time τ_{k+1}. Then for any u, $\tau_k \leq u$, we have $\mathit{FavorableToss}(r, u) = 0$. By Lemma 6, every process p that starts phase $k+1$ before time τ_{k+1} does so with x_p set to some value that p R-got in phase k, and such value can only be 0. So all reports (and thus all proposals) sent in phase $k+1$ before time τ_{k+1} are for 0. From the definition of τ_{k+1}, there are at least $n - f$ such proposals for 0 in phase $k+1$. By an argument similar to the one in the proof of Lemma 2, value 0 is $(k+2)$-locked.

Case 2.2: Now assume some value v is k-major before time τ_{k+1}.

Case 2.2.1: Suppose $v = 0$. Since 1 is never k-major, then for any time u such that $\tau_k \leq u$, we have $\mathit{FavorableToss}(r, u) = 0$. So all processes that R-get a value in phase k R-get 0. By Lemma 6, value 0 is $(k+1)$-locked.

Case 2.2.2: Now assume $v = 1$. For any time u, $\tau_k \leq u < \tau_{k+1}$, we have $\mathit{FavorableToss}(r, u) = 0$. Let S be the processes that R-get a value in phase k before time τ_{k+1}; clearly, all processes in S R-get 0.

Case 2.2.2.1: Suppose $|S| \geq n/2$. Then for any time u, $\tau_k \leq u$, $\mathit{FavorableToss}(r, u) = 0$. So, all processes that R-get in phase $k+1$ R-get 0. Note that $|S| \geq n/2$ implies that 1 can never be $(k+1)$-major. By Lemma 6, value 0 is $(k+2)$-locked.

Case 2.2.2.2: Now assume that $|S| < n/2$.

Case 2.2.2.2.1: Suppose that the coordinator of phase k does not query the R-oracle in line 20 of phase k before time τ_{k+1}. Then for any u such that $\tau_{k+1} \leq u$, we have $\mathit{FavorableToss}(r, u) = 1$. So, if the coordinator queries the R-oracle in line 20 of phase k it obtains 1 from the R-oracle. Therefore, all processes that R-get a value at or after time τ_{k+1} in phase k R-get 1. Thus, exactly $|S| < n/2$ processes R-get 0 in phase k. Since 1 is k-major, from Lemma 6 we conclude that value 0 can never be $(k+1)$-major. Since no process queries the R-oracle in phase $k+1$ before time τ_{k+1}, all processes that R-get a value in phase $k+1$ R-get 1. By Lemma 6, value 1 is $(k+2)$-locked.

Case 2.2.2.2.2: Now assume that the coordinator of phase k queries the R-oracle in line 20 of phase k before time τ_{k+1}. Then the coordinator obtains 0 from the R-oracle. So, for any $u \geq \tau_k$, we have $FavorableToss(r, u) = 0$. Since the coordinator queries the R-oracle in line 20, it received $n - f$ suggestions for ? in line 16, and this occurred before time τ_{k+1}. Thus, $n - f$ processes have set their variable x to ? in line 13 in phase k before time τ_{k+1}. Note that if any such process starts phase $k+1$, then it R-gets a value in phase k, and that value is 0, and thus such process starts phase $k+1$ with its variable x set to 0. Therefore at most $n - (n-f) = f < n/2$ processes start phase $k+1$ with their variable x set to 1. So 1 can never be $(k+1)$-major. All processes that R-get in phase $k+1$ R-get 0. By Lemma 6, value 0 is $(k+2)$-locked.□

Lemma 9 *(Termination with probability 1) If the R-oracle is random then the probability that all correct processes decide is 1.*

Proof: Immediate from Lemmata 7 and 8, and Corollary 4.□

The proof of Theorem 1 is now complete: validity and uniform agreement were shown in Corollary 3 and Corollary 2, respectively. Termination was proved in Lemma 5, and termination with probability 1 was shown in Lemma 9. $\square_{Theorem\ 1}$

From the proof of Lemma 7, it is easy to see that the expected number of rounds for termination is $O(2^{2n})$. However, it can be shown that, as in [Ben83], termination is reached in constant expected number of rounds if $f = O(\sqrt{n})$. In Section 7, we outline a similar hybrid algorithm that terminates in constant expected number of rounds even for $f = O(n)$.

6 An Optimization

The algorithm in Figure 1 was designed to be simple rather than efficient, because our main goal here is to demonstrate the viability of a "robust" hybrid approach (one in which termination can occur in more than one way: by "good" failure detection or by "good" random draws). The following optimization suggests that such hybrid algorithms can also be efficient in practice.

In many systems, failures are rare, and failure detectors can be tuned to seldom make mistakes (i.e., erroneous suspicions). The algorithm in Figure 1 can be optimized to perform particularly well in such systems. The optimized version ensures that all correct processes decide by the end of two asynchronous rounds when the first coordinator does not crash and no process erroneously suspects it.[6]

[6] Actually, decision occurs in two rounds even if up to $n - 2f - 1$ processes erroneously suspect it.

$c \leftarrow p_0$ $\{p_0$ is the first coordinator$\}$
if $p = c$ **then send** $(E, 0, v_p)$ to all processes $\{$if p is the first coordinator$\}$

wait until receive $(E, 0, v\_coord)$ **from** c **or** $c \in$*FD-oracle* $\{$query FD-oracle$\}$
if received $(E, 0, v\_coord)$
then send $(P, 0, v\_coord)$ **to** all processes
else send $(P, 0, ?)$ **to** all processes

wait for messages of the form $(P, 0, *)$ **from** $n - f$ processes $\{$"$*$" can be 0, 1 or ?$\}$
if received at least $f + 1$ $(P, 0, v)$ with the same $v \neq ?$ **then** *decide*(v)
if received at least one $(P, 0, v)$ with $v \neq ?$ **then** $x \leftarrow v$

Figure 3: Optimization for the hybrid algorithm

This optimization is obtained by inserting some extra code between lines 2 and 3 of the hybrid algorithm. This code, given in Figure 3, consists of a phase (phase 0) with two asynchronous rounds. In the first round, p_0 sends a message to all processes; in the second round, every process sends a message to all processes. We claim that: (1) the optimization code preserves the correctness of the original algorithm; and (2) processes decide quickly in the absence of failures and erroneous suspicions. To see (1) note that:

- No correct process blocks during the execution of the optimization code (phase 0), i.e., all correct processes start phase 1;
- Any process p that starts phase 1 does so with x_p set to the initial value of some process;
- If some process decides v in phase 0 then v is 1-locked. Thus, (by Corollary 4) all correct processes decide v in phase 1.

To see (2), note that if p_0 is correct and no process suspects p_0, then all processes wait for its estimate v and propose v in phase 0; so every process receives $n - f$ proposals for v and thus decides v in phase 0. Thus we have:

Theorem 2 *Theorem 1 holds for the optimized hybrid algorithm. Moreover, in executions with no crashes or false suspicions, all processes decide in two rounds.*

7 Discussion

In practice, many systems are well-behaved most of the time: few failures actually occur, and most messages are received within some predictable time. Failure-detector based algorithms (whether "pure" ones like in [CT96] or hybrid ones like in this paper) are particularly well-suited to take advantage of this: (time-out based) failure detectors can be tuned so that the algorithms perform optimally when the system behaves as predicted, and performance degrades gracefully as the system deviates from its "normal" behavior (i.e., if failures occur or messages take longer than expected). For example, the optimized version of our hybrid algorithm solves Consensus in only two asynchronous rounds in the executions that are most likely to occur in practice, namely, runs with no failures or erroneous suspicions.

The above discussion suggests that using this hybrid approach is better than using the randomized approach alone. In fact, randomized Consensus algorithms for asynchronous systems tend to be inefficient in practical settings.[7] Typically, their performance depends more on "luck" (e.g., many processes happen to start with the same initial value or happen to draw the same random bit) than on how "well-behaved" the underlying system is (e.g., on the number of failures that actually occur during execution). The fact that randomized algorithms are extremely "robust", i.e., they do not depend on how the system behaves, may also be an inherent source of inefficiency.

Note that our hybrid algorithm terminates with probability 1 even if the FD-oracle is completely inaccurate (in fact even if every process suspects every other process all the time). So it is more robust than algorithms that are simply failure-detector based.

An important remark is now in order about the expected termination time of our hybrid algorithm. We developed this algorithm by combining Ben-Or's randomized algorithm [Ben83] with the failure detection ideas in [CT96]. We selected Ben-Or's algorithm because it is the simplest, and thus the most appropriate to illustrate this approach, even though its expected number of rounds is exponential in n for $f = O(n)$. By starting from an efficient randomized algorithm, due to Chor *et al.* [CMS89], we can obtain a hybrid algorithm that terminates in constant expected number of rounds, as we now briefly explain.

Roughly speaking, the randomized asynchronous Consensus algorithm in [CMS89] is obtained from Ben-Or's algorithm by replacing each coin toss with the toss of a "weakly global coin" computed by a *coin_toss* procedure. We can do exactly the same: replace the coin tosses of the algorithm in Figure 1 with those obtained by using the *coin_toss* procedure. More precisely, in each phase, every process: (a) invokes this procedure between the second and third rounds (i.e., between lines 13 and 14) to obtain a random bit, and (b) uses this random bit rather than querying the R-oracle (in lines 20 and 25).[8]

As in [CMS89], this modified hybrid algorithm terminates[9] in constant expected number of rounds for $f \leq n(3-\sqrt{5})/2 \approx 0.38n$. But also as in [CMS89], and in contrast to the algorithm in Section 4, it assumes that the adversary cannot see the internal state of processes or the content of messages. With the optimization of Figure 3, this modified hybrid algorithm also terminates in two rounds in failure-free and suspicion-free runs.

[7] Algorithms that assume that processes *a priori* agree on a long sequence of random bits [Rab83, Tou84] are more efficient than others. But this assumption may be too strong for some systems.

[8] As in [CMS89], another simple modification is necessary: the addition of a "synchronization round" just before the *coin_toss* procedure. In this round, processes broadcast "wait" messages, then wait until $n-f$ such messages are received.

[9] Provided, of course, that the FD-oracle satisfies strong completeness.

8 Related Work

The idea of combining randomization with a deterministic Consensus algorithm appeared in [GP90], and was further developed in [Zam96]. These works, however, are for *synchronous* systems only and do not involve failure detection.

Dolev and Malki were the first to combine randomization and unreliable failure detection to solve Consensus in asynchronous systems with process crashes [DM94]. That work differs from ours in many aspects:

- The hybrid algorithms given in [DM94] assume that *both* the R-oracle and the FD-oracle always work correctly. If the failure detector loses it accuracy property, processes may decide differently; if the random source of bits is corrupted, processes may never decide.

- Two goals of [DM94] are to use failure detection to increase the resiliency and ensure the deterministic termination of randomized Consensus algorithms. The hybrid Consensus algorithms given in [DM94] achieve the first goal, by increasing the resiliency from $f < n/2$ to $f < n$, but not the second one. It is stated, however, that a future version of the paper will give an algorithm that achieves both goals.

- The two hybrid algorithms in [DM94] use failure detectors that are stronger than $\diamond S$. The first one — which supposes that the *same* sequence of random bits is shared by all the processes, as in [Rab83] — assumes that some correct process is *never* suspected by any process. The second algorithm — which drops the assumption of a common sequence of bits — assumes that $\Omega(n)$ correct processes are *never* suspected by any process.

Acknowledgement

We are grateful to Vassos Hadzilacos: some of our proofs are based on his lecture notes. We would also like to thank the anonymous referees for their valuable comments.

References

[Ben83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[BT83] Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 12–26, August 1983. An extended and revised version appeared as "Asynchronous consensus and broadcast protocols" in the *Journal of the ACM*, 32(4):824-840, October 1985.

[CD89] Benny Chor and Cynthia Dwork. Randomization in Byzantine Agreement. *Advances in Computer Research (JAI Press Inc.)*, 4:443–497, 1989.

[CHT92] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 147–158, August 1992.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4), July 1996. An earlier version appeared in [CHT92].

[CMS89] Benny Chor, Michael Merritt, and David B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM*, 36(3):591–614, 1989.

[CT91] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996. An earlier version appeared in [CT91].

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[DM94] Danny Dolev and Dalia Malki. Consensus made practical. Technical Report CS94-7, The Hebrew University of Jerusalem, March 1994.

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

[GP90] Oded Goldreich and Erez Petrank. The best of both worlds: guaranteeing termination in fast randomized Byzantine Agreement protocols. *Information Processing Letters*, 36(1):45–49, October 1990.

[Rab83] Michael Rabin. Randomized Byzantine Generals. In *Proceedings of the Twenty-Fourth Symposium on Foundations of Computer Science*, pages 403–409. IEEE Computer Society Press, November 1983.

[Tou84] Sam Toueg. Randomized Byzantine Agreements. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 163–178, August 1984.

[Zam96] Arkady Zamsky. A randomized Byzantine Agreement protocol with constant expected time and guaranteed termination in optimal (deterministic) time. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 201–208, May 1996.