Uniform Consensus is Harder than Consensus
(*extended abstract*)

Bernadette Charron-Bost* André Schiper†

Abstract

We compare the consensus problem with the uniform consensus problem in synchronous systems. In contrast to consensus, uniform consensus is not solvable in synchronous systems with byzantine failures. This still holds for the omission failure model if a majority of processes may be faulty. For the crash failure model, both consensus and uniform consensus are solvable, no matter how many processes are faulty. We consider this failure model and we examine the number of rounds required to reach a decision in consensus and uniform consensus algorithms. We show that compared with the best consensus algorithm, any uniform consensus algorithm takes at least one additional round to take a decision. We thus prove that uniform consensus is also harder than consensus whatever the failure model is.

1 Introduction

The problem of reaching agreement in a distributed system in the presence of failures is a fundamental problem of both practical and theoretical importance. One version of this problem, called *consensus*, considers that each process starts with an initial value drawn from some domain V and all non faulty processes must decide on the same value. Moreover, if the initial values are the same, say v, then the only possible decision value is v. Processors in the system are liable to fail by halting prematurely (*crash failures*), by omitting to send or receive messages when they should (*omission failures*), or by exhibiting arbitrary behaviors (*byzantine failures*).

The agreement condition of consensus, namely "no two non faulty processes decide differently", may sound odd because it allows two processes to disagree even if one of them fails a very long time after deciding. Clearly, such disagreements are undesirable in many applications since they may lead the system to inconsistent states. This is why one introduces a strengthening of the agreement condition, called the *uniform agreement* condition, which precludes any disagreement even due to faulty processes. Formally, the uniform agreement condition specifies that no two processes (whether faulty or not) decide differently. The problem that results from substituting agreement for uniform agreement is called the *uniform consensus* problem.

However, the agreement condition is not meaningless even in systems with benign failures. To see that, suppose that two processes p and q decide differently in a run of a consensus algorithm that tolerates crash failures. One of them, say p, is thereby faulty. By

*Laboratoire d'Informatique LIX, École Polytechnique, 91128 Palaiseau Cedex, FRANCE.
†Département de Systèmes de Communication, École Polytechnique Fédérale, CH-1015 Lausanne, Suisse.

a result of Charron-Bost *et al.* [CBTB00], it follows that in any system with no program-controlled crashes, when q decides, p has already crashed and q knows p to have crashed. One might take advantage of this knowledge to avoid inconsistencies resulting from disagreement. This makes the agreement condition strong enough for some distributed applications. Consequently, both consensus and uniform consensus are relevant specifications.

No matter the synchrony of the system is, the uniform agreement condition is clearly not achievable if processes may commit byzantine failures since this failure model imposes no limitation on the possible behaviors, and so on the possible decisions of faulty processes. Basically, this is the reason why agreement problems, which have been first studied in the byzantine failure model, have been originally stated with non uniform conditions [PSL80, LSP82]. Afterwards, the problem specifications that have been studied were often non uniform specifications, even in the setting of benign failures: for example, numerous results have been stated for consensus [FLP85, DDS87, DLS88, DRS90, DM90, CT96], and only a few are about uniform consensus [DS84, NT90, Lyn96]. Interestingly, Guerraoui [Gue95] showed that in most partially synchronous systems where processes may commit only crash failures, any algorithm that solves consensus also solves uniform consensus. In such systems, there is thereby no harm to solve consensus instead of uniform consensus. On the other hand, some algorithms that solve consensus in synchronous systems may violate the uniform agreement condition even for the crash failure model.

In this paper, we go over the difference between consensus and uniform consensus in the context of synchronous systems, and the differences in requirements for their solution, depending on the failure model. For byzantine and omission failures, these differences follow from classical results: In the byzantine failure model consensus has been shown to be solvable if less than one third of processes are faulty [PSL80]. As mentioned above, uniform consensus is trivially not solvable in systems with byzantine failures no matter the number of faulty processes is, and so is harder than consensus. In the omission failure model, the comparison between the two problems is far less immediate. Perry and Toueg [PT86] exhibited consensus algorithms that tolerate any number of faulty processes. For uniform consensus, we can use the translation given in [NT90] which translates any algorithm tolerant of crash failures into one tolerant of omission failures. The translation works only if a minority of processes may fail. As long as this assumption holds, any algorithm that solves uniform consensus in the crash failure model is converted by means of this translation into an algorithm that solves uniform consensus and tolerates omission failures. In systems where half or more processes may fail, Neiger and Toueg [NT90] show that uniform consensus cannot be solved with omission failures. As for the byzantine failure model, uniform consensus is therefore harder than consensus for the omission failure model because it is solvable under more restrictive conditions than consensus is.

Our own results concern the crash failure model. Both consensus and uniform consensus are solvable in this model, no matter how many processes are faulty. We show that uniform consensus is still harder than consensus by considering the time complexities of these two problems. For that, we use the well-known synchronized round model of computation that can be emulated in any synchronous system. In the presence of up to t crash failures, uniform consensus as well as consensus can be solved within $t+1$ rounds. Moreover, Lynch [Lyn96] shows that $t+1$ is a lower bound on the number of rounds required for deciding in the worst case for both of these problems. We refine this analysis by discriminating runs according to the number of failures f that actually occur: we prove that uniform consensus requires at least $f+2$ rounds whereas consensus requires only $f+1$ rounds if f is less than $t-1$. For $f = t-1$ or $f = t$, we show that both consensus and uniform consensus require $f+1$

rounds. We then present algorithms that achieve these lower bounds. Consequently, there exist early deciding algorithms for consensus where processes decide one round earlier than in any uniform consensus algorithm in most cases ($0 \leq f \leq t-2$). By refining time complexity analysis in this way, we show that uniform consensus is harder than consensus for the crash failure model.

It is important to note that contrary to [DRS90], we consider the time at which processes decide and not the time at which they halt. This is motivated by the following reasons. From a practical viewpoint, it is clearly useful to determine the time when decisions are available. On the other hand, since the $t+1$ worst case lower bound result considers the decision time and not the halting time, it seems more relevant to keep the same time complexity measure when refining efficiency analysis of consensus and uniform consensus algorithms.

The paper is organized as follows: Section 2 contains the basic definitions and the formal description of the synchronized round model of computation. Section 3 gives the number of rounds required for deciding in runs of a consensus algorithm with at most f crash failures. We investigate the same question for the uniform consensus problem in Section 4, and we prove that these lower bounds are achievable in Section 5. Section 6 provides some concluding remarks.

2 The Model

We consider synchronous distributed systems consisting of a set of n processes $\Pi = \{p_1, \cdots, p_n\}$. Processes communicate by exchanging messages. Communications are point to point. Every pair of processes is connected by a reliable channel. In such systems, one can emulate a computational model called *synchronous model* in which computations are organized in rounds of information exchange. On each process, a round consists of message sending, message receipt, and local processing. We now recall the formal description of the synchronous model [Lyn96]: Each process p_i has a buffer denoted $buffer_i$ that represents the set of messages that have been sent to p_i but that are not yet received. An algorithm A consists for each process $p_i \in \Pi$ of the following components: a set of states denoted by $states_i$, an initial state $init_i$, a message-generation function $msgs_i$ mapping $states_i \times \Pi$ to a unique (possibly *null*) message, and a state transition function $trans_i$ mapping $state_i$ and vectors (indexed by Π) of message to $states_i$. In any execution of A, each process p_i repeatedly performs the following two actions in lock-step mode:

1. Apply $msgs_i$ to the current state to generate the messages to be sent to each process. Put these messages in the appropriate buffers.

2. Apply $trans_i$ to the current state and the messages present in $buffer_i$ to obtain the new state. Remove all messages from the $buffer_i$.

The combination of these two actions is called a *round* of A. Note that in this model, an algorithm specifies the *set* of messages processes have to send in each round, but not the order in which messages are sent.

We distinguish some of the process states as *halting states*: they are those from which no further activity can occur. When reaching a halting state of algorithm A, a process stops participating to A. That is, from a halting state no messages are sent and the only state transition is a self-loop.

A *run of A* is an infinite sequence of A's rounds. A *partial run of A* is a finite prefix of a run of A.

Failures

Processes can fail by crashing, that is by stopping in the middle of their executions. A process may crash before or during some instance of the actions described above. A process may thus succeed in sending only a subset of the messages specified to be sent. This can be *any* subset since in this model, a process does not produce its messages sequentially. After crashing at a round, a process does not send any message at all the subsequent rounds.

A process is said to be *correct* in a run if it does not crash; otherwise it is said to be *faulty*. The set of all the runs of an algorithm A in which at most t processes crash is denoted by $Run(A, t)$.

Consensus and Uniform Consensus

In the consensus problem, each process starts with an input value from a fixed value set V and must reach an irrevocable decision on one value of V. The *consensus* specification is defined as the set of all the runs that satisfy the following conditions:

- *Validity:* If all processes start with the same initial value v, then v is the only possible decision value.
- *Agreement:* No two correct processes decide on different values.
- *Termination:* All correct processes eventually decide.

As explained in the Introduction, this specification allows processes to decide differently if one of them fails. To avoid such disagreements, the agreement property can be strengthened to

- *Uniform Agreement:* No two processes (whether correct or faulty) decide on different values.

The specification that results from replacing agreement by uniform agreement in the consensus specification is called *uniform consensus*. We say that algorithm A *tolerates t crashes and solves (uniform) consensus* if all the runs in $Run(A, t)$ satisfy the validity, termination , and (uniform) agreement conditions.

3 Early Deciding in Consensus

A fundamental result about consensus in synchronous systems is that if $n \geq t+2$, then any consensus algorithm that tolerates t failures must run $t+1$ rounds in some execution before processes have all decided.[1] This lower bound has been originally stated for consensus in the case of byzantine failures. The result was then extended to the case of crash failures, first for uniform consensus by Dwork and Moses [DM90], and subsequently for consensus by Lynch [Lyn96]. With respect to this worst case time complexity measure, consensus and uniform consensus are therefore two equivalent problems.

[1] Here and in the sequel, we are abusing the language slightly: by *processes have all decided* we mean that *all processes that ever decide have decided*.

This time complexity measure can be refined by discriminating runs according to the number of failures that actually occur: we consider the number of rounds required to decide not over all the runs of an algorithm that tolerates t crash failures, but over all the runs of the algorithm in which at most f processes crash for any $0 \leq f \leq t$. For consensus algorithms, Dolev, Reischuk, and Strong [DRS90] give a lower bound on the number of rounds required for processes *to halt* in the runs with at most f faulty processes. More precisely, they prove the following theorem:

Theorem 3.1 (Dolev *et al.*, **1990)** *Let A be a consensus algorithm that tolerates t process crashes. If $n \geq t+2$ then for each f, $0 \leq f \leq t$, there exists a run of A with at most f crash failures such that some process has not halted before round* $\min(t+1, f+2)$ *in that run.*

It is important to notice that a process may decide at some round without reaching a halting state, namely, it may continue to send messages and to participate to the consensus algorithm in subsequent rounds. In other words, there may be a difference between the time at which a process decides and the time at which it halts. Theorem 3.1 does not give the number of rounds until processes have all decided but rather the number of rounds until the processes all stop. Moreover, it is not clear how to derive lower bounds for deciding from lower bounds for stopping in consensus.

Nevertheless, the time at which decisions are taken is a significant time measure: from a practical view point it is indeed quite important to determine the time when decisions are available. Moreover, in order to assess the possible efficiency saving allowed by algorithms that take advantage of the actual number of failures, we need a lower bound on the number of rounds required for all processes to decide in runs with at most f failures since the $t+1$ worst case lower bound is stated in terms this measure on decision times.

In the remainder of this section, we first recall a standard consensus algorithm in which processes have all decided by the end of round $f+1$ in all the runs with at most f crash failures. This algorithm that we call *EDAC* has been originally described in [LF82]. We then easily prove that $f+1$ is indeed a lower bound for deciding in consensus algorithms.

3.1 An Early Deciding Algorithm for Consensus

In the *EDAC* algorithm, each process p_i maintains a variable *Failed* containing the set of processes that p_i detects to have crashed. Process p_i learns that p_j crashes during a round if p_i receives no message from p_j at this round. At the end of every round, each process p_i updates its variable *Failed*. If *Failed* remains unchanged during round r, that is p_i detects no new crash failure, and if p_i has not yet decided, then p_i decides at the end of round r. Any process that decides on v at round r broadcasts a (D, v) message at round $r+1$ to inform the other processes of its decision and to force the processes that have not yet decided to decide on v in turn. The code of *EDAC* is given in Figure 1.

The reader is referred to [LF82] for a correctness proof of *EDAC*. Among $f+1$ rounds of a run with at most f faulty processes, there must be some round at which no process fails. In each run of the *EDAC* algorithm with at most f crash failures, processes have thereby all decided by the end of round $f+1$ if they are still alive at this moment.

The *EDAC* algorithm proves that the $f+2$ lower bound of Theorem 3.1 does not hold when considering the question of early deciding instead of the one of early stopping. This shows that there is an actual difference between the time at which a process decides and the time at which it can halt.

states$_i$

$rounds \in N$, initially 0
$W \subseteq V$, initially the singleton set consisting of p_i's initial value
$done$, a Boolean, initially $false$
$halt$, a Boolean, initially $false$
$Rec\_Failed \subseteq \Pi$, initially $\emptyset$
$Failed \subseteq \Pi$, initially $\emptyset$
$decision \in V \cup \{unknown\}$, initially $unknown$

msgs$_i$

```
if ¬halt then
    if ¬done then send W to all processes
    else send (D, decision) to all processes
```

trans$_i$

```
if ¬halt then
    rounds := rounds + 1
    let X_j be the message from p_j, for each p_j from which a message arrives
    if done then halt := true
    if some message (D, v) arrives then
        decision := v
        done := true
    else W := W ∪ ⋃_j X_j
        Rec_Failed := Failed
        Failed := Failed ∪ {p_j : no message arrives from p_j at the current round}
        if Rec_Failed = Failed then
            decision := min(W)
            done := true
```

Figure 1: The *EDAC* algorithm

3.2 A Lower Bound for Early Deciding in Consensus Algorithms

We now prove that the *EDAC* algorithm is optimal, i.e., at least $f+1$ rounds are required for deciding in some run with at most f faulty processes. This result is a straightforward consequence of the $t+1$ worst case lower bound.

Theorem 3.2 *Let A be a consensus algorithm that tolerates t process crashes. If $n \geq t+2$ then for each f, $0 \leq f \leq t$, there exists a run of A with at most f crashes in which at least one process decides not earlier than during round $f+1$.*

Proof: Since $f \leq t$, A is a consensus algorithm that tolerates f crash failures. By the worst case lower bound, we know that there exists a run of A with at most f crashes in which some correct process decides not earlier than during round $f+1$. $\Box_{Theorem\ 3.2}$

Compared with the lower bounds stated in Theorem 3.1, this result shows that for most of the cases ($0 \leq f \leq t-2$), it takes at least one additional round to halt after deciding in early deciding algorithms.

4 Early Deciding in Uniform Consensus

We now study the question of early deciding for uniform consensus. We prove a lower bound for uniform consensus greater than the $f+1$ lower bound for consensus stated in Theorem 3.2. Since this lower bound for consensus is achievable (for example, by the *EDAC* algorithm), we thereby show that the uniform consensus problem is harder than the consensus problem, even in the context of crash failures.

Theorem 4.1 *Let A be a uniform consensus algorithm that tolerates t process crashes. For each f, $0 \leq f \leq t$, there exists a run of A with at most f crashes in which at least one process decides not earlier than during round $f+2$ if $f \leq t-2$, and not earlier than during round $f+1$ otherwise.*

Before proving the theorem, we first introduce some additional definitions and notations. Let ρ be a run of an algorithm A. For any $k \leq 1$, we define $\rho[k]$ to be the partial run of A that consists in the k first rounds of ρ. The *conservative extension* of $\rho[k]$ is the unique run ρ' of A such that $\rho'[k] = \rho[k]$ and no process crashes after round k. We say that ρ is *f-regular* if there are at most f processes that crash in ρ and for every k, $1 \leq k \leq f$, there are at most k processes that crash in $\rho[k]$. If a process crashes at round k and fails to send message m, and if no process crashes after round k, then m is said to be a *last unsent message in ρ*.

There exists a unique failure free run of A, that starts from C; this run is denoted by r_C. On the other hand, for any process p, A admits a unique run ρ_C^p starting from C and in which only p is faulty and crashes from the beginning; this run is called *the silencing of p from C*.

If ρ and ρ' are two runs of A, we say that ρ is *indistinguishable from ρ' with respect to process p*, denoted $\rho \sim_p \rho'$, if p has the same initial state and performs the same sequence of actions in ρ and ρ'.

Finally, if A solves the uniform consensus problem, then for any run ρ of A, $dec(\rho)$ denotes the unique value that is decided in ρ.

Proof: (of Theorem 4.1) Since our concern is a lower bound result, we can restrict attention to the binary uniform consensus problem, i.e., $V = \{0, 1\}$. Let A be any algorithm that solves the uniform consensus problem and that tolerates $t \geq 1$ crashes. For any integer f, $0 \leq f \leq t$, consider the set of all runs of A in which at most f processes crash. There are three cases to consider.

1. $f \in \{0, \cdots, t-2\}$. First, we use a bivalency argument borrowed from [FLP85] to show that there is an initial configuration C from which the failure free run and the silencing of some process lead to two different decision values (Lemma 4.2). We then proceed by contradiction: we first show that if in all the runs of A with at most f crashes, all processes decide by the end of round $f+1$, then any last unsent message of a f-regular run can be "added" without altering the decision value (Lemma 4.3). By successive application of this intermediate result, we obtain that all the f-regular runs starting from some initial configuration C lead to the same decision value as the failure free run r_C. In particular, any silencing of some process from C have the same decision value as r_C, contradicting the preliminary bivalent result (Lemma 4.2).

2. $f = t-1$. The case studied above provides a run with at most $t-2$ crashes (and so

with at most $t-1$ crashes), in which uniform consensus is achieved not earlier than at round $f+1=t$.

3. $f=t$. In this case, the lower bound immediately follows from the $t+1$ worst case lower bound.

$\square_{\mathit{Theorem}\ 4.1}$

Lemma 4.2 *There is an initial configuration C and there is some process p such that $dec(r_C) \neq dec(\rho_C^p)$.*

Proof: (Same as in [FLP85]). Suppose, for contradiction, that such an initial configuration does not exist. Consider the initial configurations C^0 and C^n such that all processes have initial value 0 and 1, respectively. By the validity property, C^0 is 0-valent and C^1 is 1-valent. Clearly, there are two initial configurations C and C' that differ by the initial value of only one process p and such that for any process q, $dec(r_C) = dec(\rho_C^q) = 0$ and $dec(r_{C'}) = dec(\rho_{C'}^q) = 1$. In particular, $dec(\rho_C^p) \neq dec(\rho_{C'}^p)$. Clearly, for any process $q \neq p$, we have $\rho_C^p \sim_q \rho_{C'}^p$ and thus $dec(\rho_C^p) = dec(\rho_{C'}^p)$ — a contradiction. $\square_{\mathit{Lemma}\ 4.2}$

Lemma 4.3 *Let f be an integer, $0 \leq f \leq t-2$. Suppose that in all the runs of A with at most f crashes, all the processes that are still alive at the end of round $f+1$ have decided by the end of round $f+1$. Let σ be an f-regular run of A and let m be any last unsent message of σ. If τ is the run of A which is identical to σ except that m is sent in τ, then $dec(\sigma) = dec(\tau)$.*

Proof: By definition of τ and since σ is f-regular, τ is also an f-regular run of A. Thus, after $f+1$ rounds, all the processes that are still alive have decided in both σ and τ. Note that any process is correct in τ iff it is correct in σ.

Let p be the process that fails to send m and q be the destination of m. Let k be the round of σ during which p crashes. The cases where $k > f+1$ are trivial. Thus, we assume that $k \leq f+1$. The proof is by induction on $l = f+1-k$, starting with $l=0$ and ending with $l=f$.

Basis: $l=0$, i.e., $k=f+1$. Since $n-f \geq n-(t-2) \geq 3$, there exists at least one process, different of q, that is correct in both σ and τ. For such a process s, we have $\sigma[f+1] \sim_s \tau[f+1]$. This implies that s decides the same value in σ and τ. Therefore, $dec(\sigma) = dec(\tau)$.

Inductive step: Assume $l \geq 1$. Suppose the claim is true for any last unsent message in round i of any f-regular run, with $f-l+2 \leq i \leq f+1$. Run σ is f-regular, and so there are at most k processes that crash in $\sigma[k]$. Since $k+1+l = f+2$ and $f+2 < n$, we can find l processes $r_1, \cdots, r_l$ which are different of q, and which do not crash in $\sigma[k]$. Let σ' be the run that is identical to σ, except that:

- At round $k+1$, q succeeds in sending a message only to r_1 and then crashes. No other processes fail in this round.
- At round $k+2$, r_1 succeeds in sending a message only to r_2 and then crashes. No other processes fail in this round.

  $\vdots$

- At round $f+1$, r_{l-1} succeeds in sending a message only to r_l and then crashes.
- Process r_l crashes at the beginning of round $f+2$, just before sending any message. No other processes fail in this round and in the later rounds.

Run σ is regular, and thus there are at most $k+l+1 = f+2$ crash failures in σ'. Since $f \leq t-2$, σ' is in $Run(A,t)$.

Let $\sigma^1, \cdots, \sigma^l$ denote the conservative extensions of $\sigma'[k+1], \cdots, \sigma'[f+1]$, respectively. We can safely extend this notation to $\sigma^0 = \sigma$ because m is a last unsent message of σ. Since σ is regular, there are at most $k+i$ crash failures in σ^i. In particular, there are at most f crash failures in σ^{l-1}. Process r_l is correct in σ^{l-1}, and thus decides by the end of round $f+1$ in σ^{l-1}. Moreover, σ', σ^{l-1}, and σ^l are indistinguishable to r_l up to the end of round $f+1$. This shows that process r_l decides the same value by the end of round $f+1$ in each of these three runs. Since the agreement property is uniform, this implies that

$$dec(\sigma^{l-1}) = dec(\sigma^l) = dec(\sigma'). \tag{1}$$

On the other hand, in each run σ^i, $1 \leq i \leq l$, the message that r_{i-1} fails to send to any process $s \notin \{p, q, r_1, \cdots, r_i\}$ at round $k+i$ is a last unsent message of σ^i. Moreover, $\sigma^0, \sigma^1, \cdots, \sigma^{l-1}$ are f-regular runs. In each run σ^i, the message that process r_i fails to send to any process $s \notin \{p, q, r_1, \cdots, r_{i+1}\}$ during $k+i+1$ is a last unsent message of σ^i. By successive application of the inductive hypothesis, we obtain that $dec(\sigma^{i-1}) = dec(\sigma^i)$ for any index i such that $1 \leq i \leq l-1$. Finally, this shows that

$$dec(\sigma^{l-1}) = \cdots = dec(\sigma^1) = dec(\sigma^0). \tag{2}$$

Equalities (1) and (2) imply that $dec(\sigma) = dec(\sigma')$.

Now from run τ, we use a similar construction of regular runs: let τ', and $\tau^0 = \tau, \tau^1, \cdots, \tau^l$ denote the so-defined regular runs of A. By a similar argument to those used with σ', σ^{l-1}, and σ^l, we show that

$$dec(\tau^{l-1}) = dec(\tau^l) = dec(\tau'). \tag{3}$$

By repeated applications of the inductive hypothesis, we get that

$$dec(\tau^{l-1}) = \cdots = dec(\tau^1) = dec(\tau). \tag{4}$$

This implies that $dec(\tau) = dec(\tau')$.

On the other hand, let s be a process that is correct in both σ' and τ' (such a process exists since $f+2 \leq t < n$). Runs σ' and τ' are indistinguishable to s, i.e., $\sigma' \sim_s \tau'$. This implies that $dec(\sigma') = dec(\tau')$. So $dec(\sigma) = dec(\tau)$, as needed. $\Box_{Lemma\ 4.3}$

At this point, it is interesting to compare the result stated in Theorem 4.1 with the lower bounds of [DRS90] for early stopping in consensus algorithms. Since the uniform consensus specification is stronger than the one of consensus, lower bounds for consensus hold for uniform consensus *a fortiori*. On the other hand, since a process decides before halting, lower bounds for deciding are not greater than those for stopping. This is why our lower bounds of Theorem 4.1 can neither be derived from [DRS90] nor imply those of [DRS90].

5 An Early Deciding Algorithm for Uniform Consensus

In this section, we show that the lower bounds of Theorem 4.1 are achievable in the crash failure model.

5.1 The $EDAUC$, $FloodSet$ and $Tree_t$ algorithms

The *EDAC* algorithm presented in Section 3.1 does not solve the uniform consensus problem.[2] However, it is easy to design a variant that solves uniform consensus. For that, we adapt the *EDAC* algorithm by deferring decision after broadcasting the decision value to all at the next round. This variant, called *EDAUC*, clearly achieves the lower bound of Theorem 4.1 for every f, $0 \le f \le t-2$. For $f = t$, we may consider the *FloodSet* algorithm [Lyn96] that achieves the $f+1$ lower bound of Theorem 4.1 since it solves uniform consensus within $t+1$ rounds. The remaining case $f = t-1$ is more tricky. To handle this case, we exhibit a uniform consensus algorithm $Tree_t$ that tolerates t crash failures and such that processes have all decided by the end of round t if there are less than t faulty processes.

5.2 The $Tree_t$ algorithm

The $Tree_t$ algorithm is based on the following idea: Processes $p_1, \cdots, p_{t+1}$ broadcast their initial values during the first round. Process p_j decides v_1 (p_1's initial value) if it knows that p_1 has succeeded in sending v_1 to all the processes in the first round. In general, p_j decides v_i (p_i's initial value) if p_j can decide neither v_1 nor v_2, ..., nor v_{i-1}, and p_j knows that p_i has sent its initial value to all the processes in the first round. Since at most t processes may crash, each process eventually decides some value of $\{v_1, \cdots, v_{t+1}\}$. If process p_j receives a message from p_i in the second round, p_j can safely deduce that p_i has not crashed during the first round and thus p_i has sent a message to all the processes in the first round. If that is not the case, how can p_j know whether p_i has succeeded in sending a message to all the processes in the first round? We claim that p_j needs only t rounds to determine whether p_i has failed or not in sending messages at the first round of a run in which at most $t-1$ processes crash. For this purpose, we use a strategy known as *exponential information gathering* (*EIG*, for short) defined in [BDDS87]. The basic structure used by *EIG* algorithms is a labelled tree, whose paths from the root represents chains of processes along which some values are propagated.

In the $Tree_t$ algorithm, each process maintains t *EIG* trees that are denoted by $T^1, \cdots, T^t$. Each tree T^i has $t+1$ levels, ranging from 0 (the root) to the level t (the leaves). Each node at level k, $0 \le k \le t-1$, has exactly $n-k-1$ children. Each node in T^i is labelled by a string of process indices as follows: the root is labelled by the empty string λ, and each node with label $i_1 \cdots i_k$ has $n-k-1$ children with labels $i_1 \cdots i_k\, l$ where l ranges over all the elements of $\{1, \cdots, n\} \setminus \{i_1, \cdots, i_k, i\}$. In other words, all the chains of T^i consist of distinct processes that are all different of p_i. In the course of the computation, the processes decorate the nodes of their trees with values in $\{0, 1, null\}$. Nodes at level k are decorated during the round $k+1$. Process p_j decorates the root of T^i by 1 or 0 depending on whether a message from p_i is arrived or not at p_j during the first round. The node labelled by $i_1 \cdots i_k$

[2] To see that, consider a run of *EDAC* in which all processes are correct, except p_1 and p_2, and all the initial values equal 1, except p_1's initial value that is equal to 0. Suppose p_1 crashes at the first round and succeeds in sending a message only to p_2, whereas p_2 crashes at the very beginning of round 2. Process p_2 cannot detect p_1's crash , and so decides on 0 at the end of the first round just before crashing. The other processes decide at round 3. Since they never receive p_1's initial value, they decide on 1.

in T^i is decorated by p_j with 1 if p_{i_k} has told p_j at round $k+1$ that $p_{i_{k-1}}$ has told p_{i_k} at round k that ... p_{i_1} has told p_{i_2} at round 2 that p_{i_1} has received a message from p_i at round 1. On the other hand, $i_1 \cdots i_k$ in T^i is decorated by p_j with 0 means that p_{i_k} has told p_j at round $k+1$ that $p_{i_{k-1}}$ has told p_{i_k} at round k that ... p_{i_1} has told p_{i_2} at round 2 that p_{i_1} has not received a message from p_i at round 1. Moreover, if the node labelled by $i_1 \cdots i_k$ in T^i is decorated by *null*, then it means that the chain of communication $p_{i_1}, \cdots, p_{i_k}, p_j$ has been broken by a crash failure. In round t, if process p_j detects at most $t-1$ crashes then it is allowed to decide: p_j decides v_i if 0 occurs in all the trees $T^1, \cdots, T^{i-1}$, and 0 does not occur in T^i.

Theorem 5.1 *$Tree_t$ tolerates t crashes and solves the uniform consensus problem within t rounds in all runs with at most $t-1$ faulty processes.*

The code of the $Tree_t$ algorithm and its correctness proof are given in the appendix.

5.3 A single algorithm

For each f, $0 \leq f \leq t$, we have thus exhibited a uniform consensus algorithm that achieves the corresponding lower bound of Theorem 4.1. It is possible to strengthen this result slightly by designing a single algorithm that achieves these lower bounds all at once. For that, first note $Tree_t$ achieves the lower bounds of Theorem 4.1 for both $f = t-1$ and $f = t$. The single algorithm is based on the following idea: each process runs *EDAUC* and $Tree_t$ in parallel. During the t first rounds, a process decides according to *EDAUC* if it is allowed to make a decision. Otherwise, at rounds t or $t+1$, a process decides according to $Tree_t$. At round t, an *EDAUC* decision prevails thereby over any $Tree_t$ decision. It is not hard to see that the resulting algorithm solves the uniform consensus problem, tolerates t crashes, and achieves the lower bounds of Theorem 4.1 all at once.

6 Discussion

The paper has performed an analysis of time complexities for both consensus and uniform consensus in synchronous systems with crash failures. Our analysis shows that as for both the byzantine and the omission failure models, uniform consensus is harder than consensus for the crash failure model. It is interesting to note that the difference between these two problems with crash failures lies in their time complexities, whereas the difference is already noticeable in terms of their solvabilities with byzantine or omission failures.

A result of [CBTB00] shows that any algorithm solving a problem specification also solves the uniform version of the specification in any system that is not synchronous. In particular, this implies that consensus and uniform consensus have the same complexities in such systems. Therefore, our result also shows that when they are achievable, uniformity requirements force additional costs that depend on the synchrony of the system.

Acknowledgments

We are grateful to Sam Toueg for valuable discussions and helpful comments.

References

[BDDS87] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and H. Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 42–51, August 1987.

[CBTB00] B. Charron-Bost, S. Toueg, and A. Basu. Revisiting safety and liveness in the context of failures. Technical report, LIX, École Polytechnique, January 2000.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[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.

[DM90] Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, October 1990.

[DRS90] Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

[DS84] C. Dwork and D. Skeen. Patterns of communication in consensus protocols. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 143–153, August 1984.

[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.

[Gue95] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *9th Intl. Workshop on Distributed Algorithms*, pages 87–100. Springer Verlag, LNCS 972, September 1995.

[LF82] Leslie Lamport and Michael Fischer. Byzantine generals and transaction commit protocols. Technical Report 62, SRI International, April 1982.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[NT90] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PT86] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, March 1986.

A Appendix. The $Tree_t$ algorithm

We give here the code and the correctness proof of the $Tree_t$ algorithm.

The code of the $Tree_t$ algorithm is given in Figure 2. In this algorithm, for every string x that occurs as a label of T^i, each process has a variable $val(x)^i$; the set of values that decorate T^i is denoted by $Val(T^i)$. If $X = \{val(x)^i \ : \ |x| = k-1, \ i \notin x, \ 1 \leq i \leq t\}$ arrives from p_j at round k then $update(T^1, \cdots, T^t, X)$ denotes the multiple assignment:

$$val(xj)^i := val(x)^i, \ 1 \leq i \leq t, \ |x| = k-1, \ i \notin x, \ j \notin x, \text{ and } i \neq j.$$

On the other hand, if no message arrives from p_j at round k, then $update(T^1, \cdots, T^t, null^*)$ denotes the multiple assignment:

$$val(xj)^i := null, \ 1 \leq i \leq t, \ |x| = k-1, \ i \notin x, \ j \notin x, \text{ and } i \neq j.$$

To prove that $Tree_t$ solves uniform consensus, we first give two lemmas that relate the values of the various T^i. The first lemma describes the relationships between $vals$ at different processes at adjacent levels in the trees T^k.

Lemma A.1 *After t rounds of the $Tree_t$ algorithm, for any node label y of T^k such that $val(y)^k_i \neq null$ and for any prefix xj of y, x is a node label of T^k such that $val(x)^k_j = val(y)^k_i$. In particular, $val(x)^k_j = val(xj)^k_i$.*

Proof: Obvious from the definition of the *update* procedure.

$\Box_{Lemma\ A.1}$

The second lemma describes when 0 occurs in some tree T^k.

Lemma A.2 *If 0 occurs in the set $Val(T^k)$ at any process, then p_k crashes in round 1.*

Proof: Suppose $0 \in Val(T^k)_i$, i.e, there exists a node label x of T^k such that $val(x)_i = 0$. We claim that there is some process index j such that $val(\lambda)^k_j = 0$: If $x = \lambda$ then $j = i$. Otherwise $x = i_1 \cdots i_l$ and Lemma A.1 implies that $val(\lambda)^k_{i_1} = 0$. In this case, we have $j = i_1$.

From the algorithm, $val(\lambda)^k_j = 0$ if p_k fails in sending its initial value to p_j and thus crashes during the first round.

$\Box_{Lemma\ A.2}$

The following lemma describes the set of possible decision values.

Lemma A.3 *The decision value of any process is the initial value of some process in $\{p_1, p_2, \cdots, p_{t+1}\}$.*

Proof: Suppose any process p_i decides v in round r. From the algorithm, $r = t$ or $r = t+1$.

1. $r = t$. Then p_i has received at least $n+1-t$ messages in round t and there exists an index $j \in \{1, \cdots, t\}$ such that p_i decides $v = w^j_i$.

   (a) If $1 \leq j \leq t-1$ then, from the algorithm, we have $0 \notin Val(T^j)_i$. In particular, $val(\lambda)^j_i = 1$ and w^j_i is assigned to v_j (p_j's initial value) in the first round. This shows that $v = v_j$.

(b) If $j = t$ then $0 \in Val(T^1) \cap \cdots \cap Val(T^{t-1})$. Lemma A.2 shows that $p_1, \cdots, p_{t-1}$ have crashed in the first round. Since p_i has received at least $n+1-t$ messages in round t, a message is arrived from p_t in this round and thus p_t may not have crashed during the first round. Therefore, p_i has received p_t's initial value in round 1 and $w_i^t = v_t$ at the end of the first round.

2. If $r = t+1$ there are two cases to consider:

   (a) Process p_i decides $v = decision_j$ by receiving a message $(D, decision_j)$. From the algorithm, it is clear that p_j has decided in round t. From the above case, it follows that p_j's decision value is in $\{v_1, \cdots, v_t\}$. Therefore v also belongs to $\{v_1, \cdots, v_t\}$.

   (b) Process p_i receives no $(D, decision_j)$ message in round $t+1$. In this case, p_i decides some w_i^j with $j \in \{1, \cdots, t+1\}$. There are two cases to consider:

      i. $1 \leq j \leq t$. From the algorithm, we have $0 \notin Val(T^j)$. In particular, $val(\lambda)_i^j = 1$ and w_i^j is set to v_j. Thus, p_i decides $v = v_j$.

      ii. $j = t+1$. In this case, $0 \in Val(T^1) \cap \cdots \cap Val(T^t)$. From Lemma A.2, we deduce that $p_1, \cdots, p_t$ have crashed in the first round. Since at most t processes crash, p_{t+1} is correct and has sent its initial value v_{t+1} to p_i in the first round. Therefore, p_i has set w_i^{t+1} to v_{t+1} in round 1, and thus p_i decides v_{t+1}.

$\Box_{Lemma\ A.3}$

The next two lemmas provide the key arguments to the uniform agreement property.

Lemma A.4 *If p_i decides v and p_j decides v' both in round t then $v = v'$.*

Proof: The proof is by contradiction. Suppose that in round t, p_i and p_j decide v and v', respectively, and $v \neq v'$. In this case, p_i and p_j receive at least $n+1-t$ messages in round t. From Lemma A.3, there are two indices k and l such that $v = v_k$ and $v' = v_l$. Since $v \neq v'$, we have $k \neq l$. For example, assume that $k < l$. From the algorithm, $0 \notin Val(T^k)_i$ and $0 \in Val(T^k)_j$. In other words, there exists some node label x in T^k such that

$$val(x)_i^k \neq 0,\ val(x)_j^k = 0,\ \text{and } 0 \leq |x| \leq t-1.$$

There are two cases to consider:

1. $0 \leq |x| \leq t-2$. In this case, p_j sends $val(x)_j^k = 0$ to p_i in round $|x|+2 \leq t$ and thus $val(xj)_i^k = 0$. But $0 \notin Val(T^k)_i$ — a contradiction.

2. $|x| = t-1$, i.e., there are some process indices $i_1, \cdots, i_{t-1}$ such that $x = i_1 \cdots i_{t-1}$, and so
$$val(i_1 \cdots i_{t-1})_j^k = 0 \text{ and } val(i_1 \cdots i_{t-1})_i^k \neq 0.$$
   From Lemma A.1, we have:
$$val(\lambda)_{i_1}^k = val(i_1)_{i_2}^k = \cdots = val(i_1 \cdots i_{t-2})_{i_{t-1}}^k = val(i_1 \cdots i_{t-1})_j^k = 0.$$
   Moreover, for any non empty prefix y of $i_1 \cdots i_{t-1}$, $val(y)_i^k = null$, otherwise $val(y)_i^k = val(\lambda)_{i_1}^k = 0$ — a contradiction with the fact that $0 \notin Val(T^k)_i$. In other words,
$$val(i_1)_i^k = val(i_1 i_2)_i^k = \cdots = val(i_1 \cdots i_{t-1})_i^k = null.$$

Since $val(i_1 \cdots i_{t-1})_i^k = null$ and $val(i_1 \cdots i_{t-1})_j^k = 0$, process $p_{i_{t-1}}$ crashes during round t and does not send a message to p_i in this round. In the same way, from $val(i_1 \cdots i_l)_i^k = null$ and $val(i_1 \cdots i_l)_{i_{l+1}}^k = 0$, we deduce that p_{i_l} crashes during round $l+1$ and thus no messages from p_{i_l} arrive at p_i in rounds $l+1, \cdots, t$. Moreover, p_k has crashed in the first round since it has not sent a message to p_{i_1} in this round ($val(\lambda)_{i_1}^k = 0$). Since $t \geq 2$, p_i receives no message from p_k in round t. Therefore, p_i receives no message from $p_k, p_{i_1}, \cdots, p_{i_{t-1}}$ in round t. By definition of T^k, all these processes are distinct. This yields to a contradiction with the fact that p_i receives at least $n+1-t$ messages in round t.

$\Box_{Lemma\ A.4}$

Lemma A.5 *If p_i decides v and p_j decides v' both in round $t+1$ then $v = v'$.*

Proof: The proof is similar to the proof of Lemma A.4.

$\Box_{Lemma\ A.5}$

Theorem A.6 *$Tree_t$ tolerates t crashes and solves the uniform consensus problem within t rounds in all runs with at most $t-1$ faulty processes.*

Proof:

Termination is obvious, since for any correct process p_i that does not decide in round t, $done_i = false$ in round $t+1$. Moreover, in a run with at most $t-1$ failures, any process that is still alive at the end of round t receives messages from at least $n+1-t$ processes at each round, and so takes a decision at round t.

Validity follows from Lemma A.3.

For uniform agreement, let p_i and p_j be any two processes that decide v and v' at round r and r', respectively. From the algorithm, r and r' are equal to t or $t+1$. There are two cases to consider:

1. $r = r'$. Then Lemma A.4 and Lemma A.5 imply that $v = v'$.
2. $r \neq r'$. For example, assume $r = t$ and $r' = t+1$. We consider two cases:
   (a) Process p_i is still alive when sending messages in round $t+1$. In this case, p_j receives (D, v) from p_i and thus p_j decides $v' = v$.
   (b) Process p_i does not send a message to p_j (because it crashes) in round $t+1$. Since p_j does not decide in round t, p_j receives messages from at most $n-t$ processes in this round. Process p_i is among these $n-t$ processes since it is still alive until the end of round t. Therefore, at least $t+1$ processes are crashed in round $t+1$ — a contradiction. Thus, case (b) cannot occur.

This proves that p_i and p_j make the same decision in any possible case.

$\Box_{Theorem\ A.6}$

$\mathbf{states_i}$

$rounds \in N$, initially 0 ; $T^1, \cdots, T^t$, whose all values are equal to $unknown$
$w^1, \cdots, w^{t+1} \in V \cup \{unknown\}$, initially $unknown$; $v \in V$, initially p_i's initial value
$decision \in V \cup \{unknown\}$, initially $unknown$; $done \in \{true, false\}$, initially $false$

$\mathbf{msgs_i}$

case
    $round = 0$:
        if $1 \le i \le t+1$ then send v to all processes
        else send $null$ to all processes
    $round = 1, \cdots, t-1$:
        send $\{val(x)^j \ : \ |x| = rounds - 1, \ j \notin x, \ 1 \le j \le t\}$ to all processes
    $round = t$:
        if $not(done)$ then send $\{val(x)^j \ : \ |x| = t-1, \ j \notin x, \ 1 \le j \le t\}$ to all processes
        else send $(D, decision)$ to all processes

$\mathbf{trans_i}$

$rounds := rounds + 1$
let X_j be the message from p_j, for each p_j from which a message arrives
case
    $rounds = 1$:
        for all $j \in \{1, \cdots, t+1\}$ do
            if a message is arrived from p_j then
                $w^j := X_j$
                if $j \ne t+1$ then $val(\lambda)^j := 1$
            else if $j \ne t+1$ then $val(\lambda)^j := 0$
    $rounds = 2, \cdots, t$:
        for all $j \in \{1, \cdots, n\}$ do
            if a message is arrived from p_j then $update(T^1, \cdots, T^t, X_j)$
            else $update(T^1, \cdots, T^t, null^*)$
        if $rounds = t$ then
            if at least $n+1-t$ messages are arrived then
                $done := true$
                if $0 \notin Val(T^1)$ then $decision := w^1$
                else if $0 \notin Val(T^2)$ then $decision := w^2$
                    else
                        $\ddots$
                            if $0 \notin Val(T^{t-1})$ then $decision := w^{t-1}$
                            else $decision := w^t$
    $rounds = t+1$:
        if $not(done)$ then
            if some message X_j is equal to $(D, decision_j)$ then $decision := decision_j$
            else if $0 \notin Val(T^1)$ then $decision := w^1$
                    else if $0 \notin Val(T^2)$ then $decision := w^2$
                        else
                            $\ddots$
                                if $0 \notin Val(T^t)$ then $decision := w^t$
                                else $decision := w^{t+1}$

Figure 2: The $Tree_t$ algorithm