The Time-Complexity of Local Decision in Distributed Agreement*

Partha Dutta Rachid Guerraoui Bastian Pochon
Distributed Programming Laboratory, EPFL
CH-1015, Lausanne, Switzerland

Abstract

Agreement is at the heart of distributed computing. In its simple form, it requires a set of processes to decide on a common value out of the values they propose. The time-complexity of distributed agreement problems is generally measured in terms of the number of communication rounds needed to achieve a global decision; i.e., for *all* non-faulty (correct) processes to reach a decision. This paper studies the time-complexity of *local* decisions in agreement problems, which we define as the number of communication rounds needed for *at least one* correct process to decide. We explore bounds for *early* local decision, that depends on the number f of actual failures (that occur in a given run of an algorithm), out of the maximum number t of failures tolerated (by the algorithm). We first consider the synchronous message-passing model where we give tight local decision bounds for three variants of agreement: consensus, uniform consensus and (non-blocking) atomic commit. We use these results to (1) show that, for consensus, local decision bounds are not compatible with global decision bounds (roughly speaking, they cannot be reached by the same algorithm), and (2) draw the first sharp line between the time-complexity of uniform consensus and atomic commit. Then we consider the eventually synchronous model where we give tight local decision bounds for synchronous runs of uniform consensus. (In this model, consensus and uniform consensus are similar, atomic commit is impossible, and one cannot bound the number of rounds to reach a decision in non-synchronous runs of consensus algorithms.) We prove a counter-intuitive result that the early local decision bound is the same as the early global decision bound. We also give a matching early deciding consensus algorithm that significantly improves over previous eventually synchronous consensus algorithms.

1 Introduction

Local vs Global Agreement Decisions. Determining how long it takes to reach agreement among a set of processes is an important question in distributed computing. For

*This paper is an extended and revised version of a paper titled "Tight Bounds on Early Local Decisions in Uniform Consensus" by the same authors, that appeared in the 17^{th} *International Symposium on Distributed Computing* (DISC 2003), LNCS 2848, Springer-Verlag.

instance, the performance of a replicated system is impacted by the performance of the underlying consensus service used to ensure that the replica processes agree on the same order to deliver client requests [21]. Similarly, the performance of a distributed transactional system is impacted by the performance of the underlying atomic commit service used to ensure that the database servers agree on a transaction outcome [16].

Traditionally, lower bounds on the time complexity of distributed agreement have been stated in terms of the number of communication rounds (also called communication steps) needed for *all* correct processes to decide [23] (i.e., *global decision*), or even *halt* [12], possibly as a function of the number of failures f that actually occur in a given run of an algorithm, out of the total number of failures t that are tolerated by the algorithm. (In this paper we only consider crash-stop failures.)

From a practical perspective, what we might sometimes want to measure and optimize, is the number of rounds needed for *at least one* correct process to decide, i.e., for a *local decision*. Indeed, a replicated service can respond to its clients as soon as a single replica decides on a reply (and knows that other replicas will reach the same decision). Similarly, the client of an atomic commit service might be happy to know the outcome of a transaction once the outcome has been determined, even if some database servers have yet to be informed of the outcome.

Motivations. Surprisingly, despite the large body of work on the performance of agreement, so far, no study on local decision lower bounds has appeared in the literature. To get an intuition of some of the specific ramifications underlying such a study, consider the consensus problem [28, 24] in the synchronous model, where a set of processes, $\{p_1, p_2, \ldots, p_n\}$, proceed by exchanging messages in a round by round manner, and t out of the n processes may fail by crashing [20].

The problem consists for the processes to decide on a common final value, out of values they initially proposed, such that all correct processes eventually decide and agree on a common decision. The following algorithm (from [15]) conveys the fact that there can indeed be a difference between local and global decision lower bounds. Process p_1 decides at round 0 (i.e., at the very beginning) its own initial value, and then broadcasts this value to all. Process p_2 decides at round 1, p_1's value if p_2 receives it, or otherwise its own value. At round 2, process p_3 decides p_1's value if p_3 receives it, otherwise p_2's value if p_3 receives it, otherwise its own value, and so forth. In runs of this algorithm with at most $f \leq t$ failures, at least one correct process decides by round f. Hence, if we denote by l_f the tight local decision lower bound for consensus in runs of the synchronous model with f failures, the very existence of the algorithm means that $l_f \leq f$. In fact, a closer look reveals that l_f is exactly f. However, if we denote by g_f the tight global decision lower bound, we know from [23] that g_f is exactly $f + 1$. This observation opens several questions.

- Can we match both lower bounds with the *same* algorithm? The synchronous consensus algorithm we just sketched matches the lower bound $l_f = f$ but clearly does not match the lower bound $g_f = f + 1$. Is there any other algorithm that does so? Otherwise, we would be highlighting a rather interesting trade-off in the design of consensus algorithms.

- What is the impact of the very nature of the agreement?
  - Consider for instance the *uniform* variant of consensus [18], where no process disagrees with any other process, even one that crashed. Clearly, the algorithm sketched above needs to be revisited. We can easily exhibit a uniform consensus algorithm in which at least one correct process decides by round $f+1$, in runs with at most f failures; i.e., $l_f \leq f+1$. Additionally, we know from [4, 19] that, for most values of f, $g_f = f+2$. Is $g_f = l_f + 1$?
  - Similarly, consider the non-blocking atomic commit problem [29, 18], where the processes have to decide 0 if some process proposes 0, and have to decide 1 if no process proposes 0 or crashes. We know that the tight global decision lower bound for atomic commit is the same as for uniform consensus [3, 7]. But, do the two problems have the same tight local decision bounds as well?
- What is the impact of the model? Consider consensus for instance in the eventually synchronous (ES) model [8]. If we compare (a) the number of rounds g_f^{es} needed for all correct processes to decide in *synchronous* runs with f process crashes, and (b) the number of rounds l_f^{es} needed for at least one correct process to decide in such runs, is $g_f^{es} = l_f^{es} + 1$?

Contributions.

- We show in the synchronous model that, except for some specific values of f (which we precise in the paper), $g_f = l_f + 1$ for consensus, uniform consensus and non-blocking atomic commit. (In fact, to exhibit a matching algorithm for uniform consensus and non-blocking atomic commit, we give an algorithm for the "stronger" interactive consistency problem [28].) Furthermore, we highlight an interesting trade-off in the design of consensus algorithms by showing that no consensus algorithm can match both global and local decision bounds. More precisely, no consensus algorithm can match both l_{f+1} and g_f. In addition, we show that, for the failure-free case (i.e., $f = 0$) of non-blocking atomic commit, the local decision bound is higher than that of uniform consensus. Since both problems have identical global decision lower bounds [3, 7], we draw the first line between their time-complexity.
- We also consider uniform consensus in the eventually synchronous model. (In this model, non-blocking atomic commit is not solvable when $t \geq 1$, and consensus is similar to uniform consensus [15].) We determine a local decision lower bound of $f+2$ rounds for synchronous runs of the model, with f failures (for $f \leq t-3$). Then we present an algorithm that, in synchronous runs with f failures, globally (and hence locally) decides in $f+2$ rounds. In addition to matching the local decision bound, to our knowledge, our algorithm is the first to match the $f+2$ rounds global decision lower bound presented in [4, 19, 6]. In other words, we show that, for synchronous runs of eventually synchronous model, tight local decision bounds are the same as for global decision; i.e., $g_f^{es} = l_f^{es} = f+2$.

Related Work. The consensus problem was introduced in [28, 24] and (non-blocking) atomic commit was defined in [16, 29]. The distinction between consensus and uniform consensus, and the relationship with the atomic commit problem were discussed in [18, 15]. Initial lower bound result on the time-complexity of agreement problems was proved in [13], and it was studied further in [23, 11, 27, 10, 1]. The eventually synchronous model was introduced in [9, 8].

In the synchronous model, one of the initial early halting agreement algorithms was presented in [23]. The early halting lower bound for consensus was proved in [12]. The early decision lower bound for uniform consensus, and its difference with the non-uniform case were studied in [4, 19].

In the eventually synchronous model, the first consensus algorithm was presented in [8]. The similarity between consensus and uniform consensus in the eventually synchronous model was shown in [15]. Tight bounds for synchronous runs of the eventually synchronous model, in the failure-free case ($f = 0$)was shown in [19, 31, 25], and for the worst-case ($f = t$) was shown in [6]. Techniques that use forward inductions to prove lower bounds on agreement problems were introduced in [26, 1].

In [22], Lamport pointed out *"two conflicting beliefs about the cost of making a decision"* in agreement algorithms.

- *"One is that the leader can simply decide and then inform the other processors of its decision with a single message delay [2]."*
- *"The other belief is that a three-phase commit a la Skeen [30] is needed."*

Lamport furthermore concluded that *"If making a decision is interpreted as solving the consensus problem, then neither belief is correct"*. By studying the local decision lower bound in various models and for various agreement problems, we give in this paper a precise interpretation of those beliefs that make them complementary rather than conflicting.

- We interpret the first belief as: the tight local decision lower bound for failure-free runs of consensus in a synchronous model is 0, i.e., $l_0 = 0$ for consensus in the synchronous model (the consensus algorithm sketched earlier matches this lower bound).
- We interpret the second belief as: the tight local decision lower bound for failure-free runs of non-blocking atomic commit is 2 (even in the synchronous model), i.e., $l_0 = 2$ for non-blocking atomic commit in the synchronous model.[1]

Through our interpretation of the second belief, we show here that, in a rigorous sense, non-blocking atomic commit is inherently slower than uniform consensus in synchronous model. To our knowledge, this is the first time a sharp line is drawn between non-blocking atomic commit and uniform consensus in terms of time-complexity, and this was possible precisely because we considered local decisions. In terms of global decisions, there is no difference between these problems.

[1]It is important to note, with respect to the second belief mentioned by Lamport, that the notion of phase in the parlance of Skeen [29] is different from the notion of round. The first actually captures the status of the processes (*undecided, pre-committed,* and *decided*). The decentralized three-phase commit algorithm of Skeen requires two rounds in failure-free runs [29].

Roadmap. Section 2 recalls the models we consider. Section 3 recalls the definitions of the agreement problems we study. In Section 4, we introduce the definition of our local decision metric, and we recall other time-complexity metrics. We also devise a compact notation for presenting various lower bound results on agreement problems. Section 5 recalls the layering technique of [26], also used in [19], which we slightly extend to deal with local decisions. Sections 6 and 7 present our lower bound results and matching algorithms in the synchronous model, respectively. The lower bound result for the eventually synchronous model and the matching algorithm are presented in sections 8 and 9, respectively. Section 10 concludes the paper.

2 Models

The distributed system we consider consists of a set of $n \geq 3$ processes, denoted by $\Pi = \{p_1, p_2, \ldots, p_n\}$, that communicate by message-passing: every pair of processes is connected by a bi-directional communication channel that do not duplicate or alter messages. The processes may fail by crashing and do not recover from a crash. Computation proceeds in rounds of message-exchange with round numbers starting from 1 and increasing by 1 in each successive round. A distributed algorithm A is a collection of deterministic automata, where the automata for each process executes the following two phases in every round: (a) in the *send phase*, the processes are supposed to send messages to all processes; (b) in the *receive phase*, the processes receive some messages sent in the send phase (of the current round or of a lower round) and update local states (which might include a decision event). A *run* of algorithm A is an infinite sequence of rounds of A. A *partial run* is a finite prefix of some run. A (partial) run r *extends* some partial run pr if pr is a prefix of r. Any process that does not crash in a run is said to be *correct* in that run; otherwise the process is *faulty*. We say that a message m sent in a run is *lost* (in that run) if m is never received in that run.

In the distributed system described above, a *model* is a set of runs. A *submodel* of a model M is a model that is a subset of M. We consider the following models.

- For any t such that $0 \leq t \leq n-1$, we define the t-resilient synchronous crash-stop model [20], denoted SCS_t, as follows. In any given run of SCS_t, the following four properties hold: (1) if a process starts some round k then it either completes that round or crashes; (2) at most t processes crash; (3) if a process p_i crashes in the send phase of some round k, then an arbitrary subset of messages sent by p_i in round k is received in round k, and the rest of the messages sent by p_i in round k are lost; (4) in round k, if p_i completes the send phase of the round, then every process that completes the receive phase of the round, receives the round k message sent by p_i.

- For any t such that $0 \leq t \leq n-1$, we define the t-resilient eventually synchronous crash-stop model, denoted ES_t, as follows. In ES_t, the runs may be "asynchronous" for an arbitrary yet finite number of rounds but eventually become "synchronous." A message sent in the "asynchronous period" may be *delayed* for a finite number of rounds; i.e., received in a round higher than the round in which it was sent. More precisely, in any given run of ES_t, the following four properties hold: (1) if a process starts some round k then it either completes that round or crashes; (2) (*t-resilience*) at most t processes crash, and every process which completes any round k, receives the round k messages from at least $n-t$ processes; (3) (*reliable channels*) any message sent by a correct process to a correct process in any round k is received in round k or a higher round;[2] (4) (*eventual synchrony*) there is a unknown but finite round number GSR (Global Stabilization Round) such that, in every round $k \geq GSR$, (4.1) if p_i crashes in the send phase of round k, then an arbitrary subset of messages sent by p_i in round k is received in that round, and the rest of the messages sent by p_i in round k is lost, and (4.2) if p_i completes the send phase of the round k, then every process that completes the receive phase of the round, receives the round k message sent by p_i. (We say that the run is *synchronous from round GSR*.)

Observe that, for any $0 \leq f \leq t \leq n-1$, SCS_f is a submodel of SCS_t, and ES_f is a submodel of ES_t. Furthermore, every run of SCS_t is a run of ES_t with $GSR = 1$. Thus, SCS_t is a submodel of ES_t.

Hereafter, we make a slight change in terminology: we say that each of the n models SCS_t $(0 \leq t \leq n-1)$ is a different synchronous model (i.e., there is no unique synchronous model). Similarly, we say that each of the n models ES_t $(0 \leq t \leq n-1)$ is a different eventually synchronous model.

3 Agreement Problems

We consider three agreement problems: consensus, uniform consensus and non-blocking atomic commit.

- In the (non-uniform) *consensus* problem [24], denoted NC, the processes start with a proposal value and are supposed to eventually decide on a final value such that the following properties are satisfied: (validity) if a process decides v, then some process has proposed v; (agreement) no two correct processes decide differently; and (termination) every correct process eventually decides.

- *Uniform consensus* [18], denoted UC, is a variant of consensus in which the agreement property is replaced by the following *uniform agreement* property: no two processes decide differently.

- In the *non-blocking atomic commit* problem [16, 29], denoted NBAC, each process is supposed to cast a vote of whether to abort or commit a transaction, and eventually decide. The termination and the agreement properties are the same as that for uniform consensus. Validity is defined in two parts: (abort validity) abort can be decided only if some process proposes to abort or fails, and (commit validity) commit can be decided only if all processes propose to commit. For presentation uniformity, we make the following changes in notation: (1) we say that a process proposes 0 (resp. 1) if the

[2] In other words, messages from correct to correct processes are never lost, but may be delayed by an arbitrary yet finite number of rounds.

process votes abort (resp. commit), and (2) we say that a process decides 0 (resp. 1) if the process decides to abort (resp. to commit).

To prove our lower bounds, we introduce variants of consensus and uniform consensus. We define the *weak binary agreement* problem, denoted WA, where the processes are allowed to propose either 0 or 1. WA satisfies the agreement and termination properties of consensus, and the following modified validity property (from [19]): for every value $v \in \{0, 1\}$, there is a failure-free run in which correct processes decide v. The *weak binary uniform agreement* problem, denoted UA, is identical to WA except that it also satisfies uniform agreement (no two processes decide differently).

Clearly, any NC, UC or NBAC algorithm can solve WA without any extra communication. Thus, our time-complexity lower bounds on WA immediately apply to the three agreement problems. Similarly, any UC or NBAC algorithm can solve UA without any extra communication, and hence, our time-complexity lower bounds on UA immediately apply to UC and NBAC problem.

In the synchronous model, we present the matching algorithms for uniform consensus and non-blocking atomic commit by first devising an interactive consistency algorithm, which we then transform to consensus and non-blocking atomic commit algorithms. In the *Interactive Consistency* problem [28], denoted IC, each process is supposed to propose an initial value and eventually decide on a vector of values. Termination and agreement are the same as for uniform consensus. Validity is defined as follows: for any decision vector V, the j^{th} component of V is either the value proposed by p_j or $\perp$, and may be $\perp$ only if p_j fails.

4 Time Complexity Metrics

Let r be a run of an algorithm that solves one of the agreement problems described in Section 3. We say that a process p_i *decides* in round $k \geq 1$ in r if p_i decides in the receive phase of round k, and a process decides at round 0 if it decides before sending any message in round 1. We say that a process *halts* in round k in r if it does not crash by round k, and does not take any step after round k.

We distinguish four different time complexity metrics for runs of agreement algorithms: *global decision*, *global halting*, *local decision* and *local halting*. Consider any run r of an algorithm that solves an agreement problem.

- We say that run r *globally decides* (resp. *globally halts*) in round k if all correct processes decide (resp. halt) in round k, or in a *lower* round, and some correct process decides (resp. halts) in round k [13, 12, 4, 19].

- We say that run r *locally decides* (resp. *locally halts*) in round k if all correct processes decide (resp. halt) in round k, or in a *higher* round, and some correct process decides (resp. halts) in round k.

We introduce the following notations. If a run r globally decides at round k, we write $(r, gd) = k$. Similarly, the round at which r globally halts, locally decides, and locally halts, are denoted by (r, gh), (r, ld), (r, lh), respectively. Note that, since every correct process decides before it halts, $(r, ld) \leq (r, lh)$, and $(r, ld) \leq (r, gd) \leq (r, gh)$. Given a model M1, a submodel M2 of M1, an agreement problem P, and a time complexity metric T, we denote by the ordered tuple (M1, M2, P, T) the following tight bound. (M1, M2, P, T) is the round number k such that (1) (lower bound) every algorithm that solves P in M1 has a run r in M2 such that $(r, T) \geq k$, and (2) (matching algorithm) there is an algorithm Alg that solves P in M1 such that, every run r of Alg in M2 has $(r, T) \leq k$. In other words, given a problem P and a time-complexity metric T, (M1, M2, P, T) is the tight bound for achieving T, on those runs in M2, that are runs of algorithms that solve P in M1.

Before delving into our lower bounds, we recall some known results on consensus (NC) and uniform consensus (UC) using our notation. (For every pair of reals $a \leq b$, $[a, b]$ denotes the set of integers x such that $a \leq x \leq b$.)

- $\forall t \in [0, n-2]$, (SCS_t, SCS_t, NC, gd) $= t+1$. Every consensus algorithm in SCS_t has a run (in SCS_t) in which some correct process decides in round $t+1$ or in a higher round, and there is a consensus algorithm A in SCS_t such that, in every run of A (in SCS_t), every correct process decides by round $t+1$ [13, 20].

- $\forall t \in [2, n-1]$, $\forall f \in [0, t-2]$, (SCS_t, SCS_f, NC, gh) $= f+2$. Every consensus algorithm in SCS_t has a run in SCS_f in which some correct process halts in round $f+2$ or in a higher round, and there is a consensus algorithm A in SCS_t such that, in every run of A in SCS_f, every correct process halts by round $f+2$ [12].

- $\forall t \in [2, n-1]$, $\forall f \in [0, t-2]$, (SCS_t, SCS_f, UC, gd) $= f+2$. Every uniform consensus algorithm in SCS_t has a run in SCS_f in which some correct process decides in round $f+2$ or in a higher round, and there is a uniform consensus algorithm A in SCS_t such that, in every run of A in SCS_f every correct process decides by round $f+2$ [4, 19].

- $\forall t \in [1, (n-1)/2]$, ($ES_t$, SCS_t, NC, gd) $= t+2$. Every consensus algorithm in ES_t has a run in SCS_t in which some correct process decides in round $t+2$ or in a higher round, and there is a consensus algorithm A in ES_t such that, in every run of A in SCS_t every correct process decides by round $t+2$ [6].

Roughly speaking, in this paper we investigate tight bounds when the time-complexity metric is local decision (ld). In particular, we determine (SCS_t, SCS_f, UC, ld), (SCS_t, SCS_f, NC, ld), and (ES_t, SCS_f, UC, ld).

5 Layering

Our lower bound proofs are devised following the layering technique of [26], also used in [19]. We first introduce some definitions and then recall the notion of layering from [26, 19]. We then present two lemmas (that are slightly modified from [19]) from which we derive our lower bound results. (In the following, we point out when our notions differ from [19].)

5.1 Configurations and extensions

Consider a model M and an agreement algorithm A devised in M. For each run r of algorithm A in model M, we denote by $val(r)$ the decision value of any correct process in r. (This definition is unambiguous because, in every agreement problem we consider, no two correct processes decide differently.) For any run r of A in M we define the *configuration* C at the end of round k (also called *round k configuration*), as an ordered tuple of size $n + n^2$, where the element i, for $1 \leq i \leq n$, is the state of process p_i at the end of round k in run r, and the rest of the elements contain the set of delayed messages in the n^2 communication channels at the end of round k in run r: the delayed messages in the channel from p_a to p_b is contained in element $an + b$.[3] The state of a process which has crashed is denoted by special symbol $\perp$. We say that a process p_i is *alive* in a configuration if p_i has not crashed in that configuration. In the *initial configuration* (which we also call round 0 configuration) of run r, the state of each process is its proposal value, and the state of every communication channel is the emptyset $\emptyset$.

Given a round k configuration C of algorithm A in model M, we define the following concepts. A run r of algorithm A in model M is an *extension* of the round k configuration C if the round k configuration of run r is C. A round k_1 configuration C$'$ of algorithm A in model M is an *extension* of the round k configuration C if $k \leq k_1$ and there is a run r of A in M such that the round k configuration of r is C and round k_1 configuration of r is C$'$. If M is a synchronous model, we denote by $r(C)$ the run which is an extension of C such that no process crashes after round k. We define $val(C)$ as $val(r(C))$. Observe that a process p_i is alive in C if and only if p_i is correct in $r(C)$.

5.2 Layering in synchronous models

For every t such that $0 \leq t \leq n-1$, we define SCS1$_t$ as the submodel of SCS$_t$ that contains those runs of SCS$_t$ in which at most one process crashes in a round. In the rest of this subsection, we consider any given weak binary agreement (WA) algorithm A in model $SCS1_t$. (See Section 3 for a definition of WA.)

A run of algorithm A is completely defined by its initial configuration and its failure pattern. (The failure pattern for a run in $SCS1_t$ consists, for each round k, of the process p_i which crashes in round k, and the set of processes which did not receive the round k message from p_i.) We denote an extension by one round, of a round k configuration C, as follows: for $1 \leq i \leq n$ and $S \subseteq \Pi$, $C.(i, S)$ denotes the round $k+1$ configuration reached by crashing p_i in round $k+1$ such that a process p_j *does not* receive a round $k+1$ message from p_i if any of the following holds: (1) $p_j = p_i$, (2) p_j is crashed in C, or (3) $p_j \in S$. Configuration $C.(0, \emptyset)$ denotes the one round extension of C in which no process crashes. Clearly, $C.(i, S)$ for $i > 0$ and $S \subseteq \Pi$, is a possible extension of C if at most $t-1$ processes have crashed in C and p_i is alive in C $-$ we then say that (i, S) is *applicable* to C. Configuration $C.(0, \emptyset)$ is always applicable to C.

A layer $L(C)$ is the set of configurations defined as $\{C.(i, S) | i \in \Pi, S \subseteq \Pi, (i, S)$ is ap-

[3]Since there are no delayed messages in synchronous models, the channels are empty at the end of any round. Hence, in a synchronous model, we ignore state of channels in configurations at the end of any round.

plicable to $C\}$. (In other words, if C is a round k configuration, then $L(C)$ is the set of all round $k+1$ configuration that extends C in $SCS1_t$.) For a set of round k configurations SC, $L(SC)$ is a set of round $k+1$ configurations defined as $\cup_{C \in SC} L(C)$. $L^k(SC)$ is recursively defined as follows: $L^0(SC) = SC$ and for $k > 0$, $L^k(SC) = L(L^{k-1}(SC))$. Two configurations C and D at the same round are *similar*, denoted $C \sim D$, if they are identical or they differ at exactly one process.[4]

A set of configurations SC is *similarly connected* if, for every pair of configurations $C, D \in SC$, there are configurations $C = C_0, \ldots, C_m = D$ such that $C_i \sim C_{i+1}$ for every i such that $0 \leq i \leq m-1$. We now revisit Lemma 2.3 of [19]. Roughly speaking, this lemma says that, in $SCS1_t$, if we start with a similarly connected set SC of configurations, we can keep the set of extensions from SC similarly connected, provided we can crash one process in every round we extend.

Lemma 1 *Let $SC = L^0(SC)$ be a similarly connected set of configurations such that in any configuration of SC no process has crashed, then for all $k \in [1, t]$, $L^k(SC)$ is a similarly connected set of configurations in which no more than k processes are crashed in any configuration.*

Proof: (A simple modification of the proof of [19].[5]) The proof is by induction on round number k. The base case $k = 0$ is immediate. For the inductive step, assume that $L^{k-1}(SC)$ is similarly connected and in every configuration of $L^{k-1}(SC)$ at most $k-1$ processes have crashed. Notice that, in every extension by one round which is applicable to a configuration in $L^{k-1}(SC)$, at most one more process can crash. Therefore, in every configuration in $L^k(SC)$ at most k processes have crashed. We show the following two claims, which immediately imply that $L^k(SC)$ is similarly connected: (1) for any configuration $C \in L^{k-1}(SC)$, $L(C)$ is similarly connected, and (2) for any pair of configurations $C, D \in L^{k-1}(SC)$, if $C \sim D$ then $L(C) \cup L(D)$ is similarly connected.

1. Consider any configuration in $L(C)$ that is different from $C.(0, \emptyset)$, say $C1 = C.(i, S1)$, where $S1 \subseteq \Pi$, and p_i is alive in C. We claim that $C1$ and $C.(0, \emptyset)$ are similarly connected. Since $C1$ is arbitrarily selected from $L(C)$, our claim implies that every configuration in $L(C)$ is similarly connected to $C.(0, \emptyset)$, and hence, $L(C)$ is similarly connected.

   Now we prove our claim. $C.(i, \emptyset) \sim C.(0, \emptyset)$ since the configurations only differ at p_i. If $S1 = \emptyset$ then we are done. Hence, let $S1 = \{q_1, q_2, \ldots, q_m\}$. For every l in $[1, m]$, let $S1_l = \{q_1, \ldots, q_l\}$, and $S1_0 = \emptyset$. For every l in $[0, m-1]$, $C.(i, S1_l) \sim C.(i, S1_{l+1})$ because the two configurations differ only at q_{l+1}. Thus, $C(i, \emptyset) = C.(i, S1_0)$ and $C1 = C.(i, S1_m)$ are similarly connected.

[4] Our definition of similarity does not include the second requirement of the original definition in [19]: there exists a process that is alive in both C and D, and has identical states in C and D.

[5] The statement of the Lemma is similar, however, the proof is slightly different because our model $SCS1_t$ is slightly different from that of [19] - their model is actually a submodel of $SCS1_t$ in which messages sent from a crashed process are lost by only a prefix of processes in $\Pi = \{p_1, ..., p_n\}$.

2. We only need to show that if $C \sim D$ and $C, D \in L^{k-1}(SC)$ then there are configurations $C' \in L(C)$ and $D' \in L(D)$ which are similar. Let p_i be the process such that C and D are different only at p_i. (If C and D are identical then the claim is trivially true.) Then, configurations $C.(i, \Pi)$ and $D.(i, \Pi)$ are identical because no process receives message from p_i in round $k+1$. Hence, $C.(i, \Pi) \sim D.(i, \Pi)$.

□

Roughly speaking, the next lemma says that, for any WA algorithm in $SCS1_t$, there are two f round configurations which are almost identical (differ at only one process) but have different decision values in failure-free extensions. We use this lemma frequently in our lower bound proofs.

Recall that, for any configuration y in a synchronous model, $val(y)$ is the decision value of correct processes in a run which extends y and has no crashes after y.

Lemma 2 *Consider any WA algorithm A in $SCS1_t$ such that $t \in [1, n-1]$. For every $f \in [0, t]$, there are two runs of A in $SCS1_t$ such that their round f configurations, y and y', satisfy the following: (1) at most f processes have crashed in each configuration, (2) the configurations differ at exactly one process, and (3) $val(y) = 0$, whereas $val(y') = 1$.*

Proof: Consider any initial configuration C′ of algorithm A in model $SCS1_t$. Let C be the configuration in which all processes propose 0. Consider the following $n-1$ (not necessarily distinct) initial configurations: for every i in $[1, n-1]$, in configuration C_i, processes p_1 to p_i propose the same value as in C′, and the remaining processes propose 0. Notice that, for every i in $[1, n-2]$, C_i and C_{i+1} may differ only at p_{i+1}. Furthermore, C_1 and C may differ only at p_1, and C′ and C_{n-1} may differ only at p_n. Thus C and C′ are connected through a chain of configurations, any two adjacent configurations in which are similar. Since C′ was arbitrarily selected, the set of initial configurations of A in $SCS1_t$ is similarly connected. From Lemma 1 it follows that, the set of round f configurations of A in $SCS1_t$ is similarly connected.

Consider any failure-free run $r0$ of algorithm A in which correct processes decide 0. (From the validity property of WA, such a run of A exists.) We denote by y, the round f configuration of $r0$. Similarly, consider any failure-free run $r1$ of A in which correct processes decide 1. We denote by y', the round f configuration of $r1$. Obviously, $val(y) = 0$ and $val(y') = 1$.

As the set of round f configurations of A in $SCS1_t$ is similarly connected, there are some round f configurations of A in $SCS1_t$, $y = y_0, y_1, \ldots, y_m = y'$, such that $y_j \sim y_{j+1}$ for every j in $[0, m-1]$. Clearly, there is some $y_i \in \{y_0, \ldots, y_{m-1}\}$ such that, $val(y_0) = \ldots = val(y_i) \neq val(y_{i+1})$. (Otherwise, $val(y) = val(y_0) = val(y_1) = \ldots = val(y_m) = val(y')$; a contradiction.)

As $val(y_i) = val(y_0)$ and $y = y_0$, $val(y_i) = 0$. Therefore, $val(y_{i+1}) = 1$. Since both y_i and y_{i+1} are round f configurations in $SCS1_t$, at most f processes have crashed in each configuration. As $y_i \sim y_{i+1}$, the two configurations are either identical or differ at exactly one process. Since, $val(y_i) \neq val(y_{i+1})$, the configurations cannot be identical, i.e., they differ at exactly one process. □

6 Synchronous Lower Bounds

6.1 Uniform Consensus

In the following, we give two lower bounds for weak binary uniform agreement (UA) in the synchronous models (SCS_t with $1 \le t \le n-1$). Since any uniform consensus (UC) and non-blocking atomic commit (NBAC) algorithm solves UA, the lower bounds immediately apply to UC and NBAC.

We observe that any UA algorithm also solves WA, and every run of an algorithm in $SCS1_t$ is also a run in SCS_t. Thus, Lemma 2 holds when WA and $SCS1_t$ are replaced with UA and SCS_t, respectively.

Local decision. The following proposition says that any UA algorithm in SCS_t has a run in SCS_f (i.e., a run with f crashes) in which every correct process decides in round $f+1$ or a higher round.

Proposition 3 *$\forall t \in [1, n-1]$, $\forall f \in [0, t-1]$, (*SCS$_t$*,* SCS$_f$*, UA, ld) $\geq f+1$.*

Proof: Suppose by contradiction that there is a UA algorithm A in SCS_t and an integer f in $[0, t-1]$ such that, in every run of A with f failures, some correct process decides by round f.

From Lemma 2, it follows that there are two runs of A in SCS_t such that their round f configurations, y and y', satisfy the following: (1) at most f processes have crashed in each configuration, (2) the configurations differ at exactly one process, say p_i, and (3) $val(y) = 0$ and $val(y') = 1$.

From our initial assumption about algorithm A, it follows that there is an alive process q_1 in y which has already decided. (Otherwise, since every correct process in $r(y)$ is an alive process in y, $r(y)$ is a run with at most f crashes in which no correct process decides by round f.) Furthermore, q_1 has decided $val(y) = 0$ in $r(y)$ (and hence, in y) because q_1 is a correct process in $r(y)$. Similarly, in y', there is an alive process q_2 which has decided $val(y') = 1$. There are two cases to consider.

(1) $q_1 \neq p_i$: As y and y' are identical at all processes different from p_i, in y', q_1 is alive and has decided 0. Thus in $r(y')$, q_1 is a correct process and decides 0. However, in $r(y')$ every correct process decides $val(y') = 1$; a contradiction.

(2) $q_1 = p_i$: We distinguish two subcases:

- $q_2 = p_i$: Thus $p_i = q_1 = q_2$, and hence, p_i is alive in y and y'. Consider a run $r1$ which extends y and in which p_i crashes before sending any message in round $f+1$. (Recall that $f \le t-1$.) As p_i has decided 0 in y, from uniform agreement, it follows that every correct process decides 0 in $r1$. Since $t < n$, there is at least one correct process, say p_l in $r1$. Now consider a run $r2$ which extends y' and in which p_i crashes before sending any message in round $f+1$. Notice that no correct process can distinguish $r1$ from $r2$: at the end of round f no alive process which is distinct from p_i can distinguish y

from y', and p_i crashes before sending any message in round $f+1$. Thus every correct process decides the same value in $r1$ and $r2$, in particular p_l decides 0 in $r2$. However, $p_i = q_2$ decides 1 in $r2$; a contradiction with uniform agreement.

- $q_2 \neq p_i$: Then, q_2 has the same state in y and y'. Thus in y, q_2 is alive and has decided 1. In any run which extends y, $p_i = q_1$ has decided 0 and q_2 has decided 1; a contradiction with uniform agreement.

□

C-decision. The above proposition gives a lower bound on the number of rounds required for *at least one* correct process to decide (local decision). The global decision lower bound of [4, 19] determines the necessary number of rounds required for *all* correct processes to decide. It is natural to seek for the number of rounds required when we consider an intermediate time complexity metric for the runs.

For every c in $[1, n]$, we say that a run r of an agreement algorithm *c-decides* in round k, and we write $(r, d_c) = k$, if at least c correct processes decide by round k and less than c correct processes decide before round k. Using this notation, in a run with f crashes, the local decision is 1-decision, and the global decision is $(n-f)$-decision.

In the following proposition we state that any UA algorithm in SCS_t has a run in SCS_f (i.e., a run with f crashes) in which *at most one* correct process decides in round $f+1$ or in a *lower* round. In other words, the number of rounds needed for c-decision, when $c \geq 2$, is $f+2$. (In this terminology, Proposition 3 states that the number of rounds needed for 1-decision is $f+1$.)

Proposition 4 $\forall t \in [3, n-1]$, $\forall f \in [0, t-3]$, $\forall c \in [2, n-f]$, *(*SCS_t, SCS_f, *UA*, d_c*)* $\geq f+2$.

Proof: It is sufficient to show that (SCS_t, SCS_f, UA, d_2) $\geq f+2$. Suppose by contradiction that there is an UA algorithm A in SCS_t and an integer f in $[0, t-3]$ such that, in every run of A with at most f failures, at least two correct processes decide by round $f+1$.

From Lemma 2 it follows that there are two runs of A in SCS_t such that their round f configurations, y and y', satisfy the following: (1) at most f processes have crashed in each configuration, (2) the configurations differ at exactly one process, say p_i, and (3) $val(y) = 0$ and $val(y') = 1$. Let z and z' denote the configurations at the end of round $f+1$ of $r(y)$ and $r(y')$, respectively.

From our initial assumption about algorithm A, it follows that, in z, there are two alive processes q_1 and q_2 which have decided $val(y) = 0$. (Otherwise, if at most one alive process has decided in z, then at most one correct process has decided by round $f+1$ in $r(y)$, a run with f crashes; a contradiction.) Similarly, in z', there are two alive processes q_3 and q_4 which have decided $val(y') = 1$. Since q_1 and q_2 are distinct, at least one of them is distinct from p_i, say q_1. Similarly, without loss of generality we may assume that q_3 is distinct from p_i.

Thus, we have (1) a round $f+1$ configuration z and a process q_1 such that at most f processes have crashed in z, and q_1 is alive and has decided 0 in z, (2) a round $f+1$ configuration z' and a process q_3 such that at most f processes have crashed in z', and q_3 is alive and has decided 1 in z', and (3) process p_i is distinct from both q_1 and q_3. (Processes q_1 and q_3 might not be distinct.) There are two cases to consider.

Case 1. Process p_i is alive in y and y'. Consider the following two runs of A:
R1 is a run such that (1) the round f configuration is y, (2) p_i crashes in the send phase of round $f+1$ such that only q_1 receives the message from p_i, (3) q_1 and q_3 crash before sending any message in round $f+2$, and (4) no process distinct from p_i, q_1, and q_3 crashes after round f. Notice that q_1 cannot distinguish the round $f+1$ configuration of $R1$ from z, and therefore, decides 0 at the end of round $f+1$ in $R1$. By uniform agreement, every correct process decides 0. Since $t \leq n-1$, there is at least one correct process in $R1$, say p_l.
R2 is a run such that (1) the round f configuration is y', (2) p_i crashes in the send phase of round $f+1$ such that only q_3 receives the message from p_i, (3) q_1 and q_3 crash before sending any message in round $f+2$, and (4) no process distinct from p_i, q_1, and q_3 crashes after round f. Notice that q_3 cannot distinguish the round $f+1$ configuration of $R2$ from z', and therefore, decides 1 at the end of round $f+1$ in $R2$. However, p_l cannot distinguish $R1$ from $R2$: at the end of round $f+1$, the two runs are different only at p_i, q_1, and q_3, and none of the three processes sends messages after round $f+1$ in both runs. Thus (as in $R1$) p_l decides 0 in $R2$; a contradiction with uniform agreement.

Case 2. Process p_i has crashed in either y or y'. (Process p_i has not crashed in both y and y' because p_i has different states in y and y'.) Without loss of generality, we can assume that p_i has crashed in y, and hence, p_i is alive in y'. Consider the following two runs of A:
R12 is a run such that (1) the round f configuration is y (and hence, p_i has crashed before round $f+1$), (2) no process crashes in round $f+1$, (3) q_1 and q_3 crash before sending any message in round $f+2$, and (4) no process distinct from p_i, q_1 and q_3 crashes after round f. Observe that the round $f+1$ configuration of $R12$ is z, and hence, q_1 decides 0 at the end of round $f+1$ in $R12$. Due to uniform agreement, every correct process decides 0 in $R12$. Since $t \leq n-1$, there is at least one correct process in $R12$, say p_l.
R21 is a run such that (1) the round f configuration is y', (2) p_i crashes in the send phase of round $f+1$ such that only q_3 receives the message from p_i, (3) q_1 and q_3 crash before sending any message in round $f+2$, and (4) no process distinct from p_i, q_1 and q_3 crashes after round f. Notice that q_3 cannot distinguish the round $f+1$ configuration of $R21$ from z' because it receives the round $f+1$ message from p_i in both runs. Thus (as in z') q_3 decides 1 at the end of round $f+1$ in $R21$. However, p_l cannot distinguish $R12$ from $R21$: at the end of round $f+1$, the two runs are different only at p_i, q_1 and q_3, and none of them sends messages after round $f+1$ in both runs. Thus (as in $R12$), p_l decides 0 in $R21$; a contradiction with uniform agreement. □

Remark: Since in a run with f failures, the global decision is $(n-f)$-decision, Proposition 4 generalizes the $f+2$ round global decision lower bound of [4, 19]. However, the values of f and t, for which our result is valid are slightly different from [4, 19], e.g., unlike the lower bound of [4, 19], Proposition 4 does not consider the case where $f = t-2$ as well as the case where $t = 2$.

6.2 Consensus

We give here two lower bounds for weak binary agreement (WA) in synchronous models (SCS_t with $1 \leq t \leq n-1$) that correspond to Proposition 3 and Proposition 4. We then show the impossibility of simultaneously matching both local decision and global decision lower bounds of WA. Since any consensus (NC) algorithm solves WA, the results immediately apply to consensus.

We observe that every run of an algorithm in $SCS1_t$ is also a run in SCS_t. Thus, Lemma 2 holds when $SCS1_t$ is replaced with SCS_t.

Local decision. The following proposition states that any WA algorithm in SCS_t has a run in SCS_f (i.e., a run with f crashes) in which every correct process decides in round f or a higher round.

Proposition 5 $\forall t \in [1, n-1]$, $\forall f \in [0, t]$, $(\mathrm{SCS}_t, \mathrm{SCS}_f, WA, ld) \geq f$.

Proof: Suppose by contradiction that there is an WA algorithm A in SCS_t and an integer f in $[0, t]$ such that, in every run of A with f failures, some correct process decides by round $f-1$. Notice that the contradiction is immediate for the case $f = 0$: no process can decide by round -1. So we consider the case $f \in [1, t]$.[6]

From Lemma 2 it follows that there are two runs of A in SCS_t such that their round $f-1$ configurations, y and y', satisfy the following: (1) at most $f-1$ processes have crashed in each configuration, (2) the configurations differ at exactly one process, say p_i, and (3) $val(y) = 0$ and $val(y') = 1$.

From our initial assumption about algorithm A, it follows that there is an alive process q_1 in y which has already decided. (Otherwise, since every correct process in $r(y)$ is an alive process in y, $r(y)$ is a run with at most $f-1$ crashes in which no correct process decides by round $f-1$.) Furthermore, q_1 has decided $val(y) = 0$ in $r(y)$ (and hence, in y) because q_1 is a correct process in $r(y)$.

We now show that no alive process distinct from p_i has decided in y (which implies $p_i = q_1$). Suppose by contradiction that some alive process distinct from p_i, say q_2, has decided in y. Since q_2 is alive in y, it is correct in $r(y)$, and hence, has decided $val(y) = 0$ in y. Since y and y' differ only at p_i, and p_i is distinct from q_2, q_2 is alive and has decided 0 in y'. Thus, in $r(y')$, q_2 is a correct process and decides 0. However, every correct process in $r(y')$ decides $val(y') = 1$; a contradiction.

Thus, p_i is the only alive process that has decided in y. Consider any run r' that extends y and in which only process p_i crashes after round $f-1$. At most f processes crash in r'. At the end of round $f-1$ in r', the only alive process which has decided is p_i, but p_i is a faulty process in r'. Thus, r' is a run with f failures in which no correct process decides by round $f-1$; a contradiction. □

C-decision. The next proposition states that any WA algorithm in SCS_t has a run in SCS_f (i.e., a run with f crashes) in which *at most one* correct process decides in round f or in

[6] Recall that, we define deciding at round 0, as deciding before sending any message in round 1.

a *lower* round. In other words, the number of rounds needed for c-decision, when $c \geq 2$, is $f+1$. (Proposition 5 states that the number of rounds needed for 1-decision is f.)

Proposition 6 $\forall t \in [1, n-1]$, $\forall f \in [0, t]$, $\forall c \in [2, n-f]$, *(*SCS_t, SCS_f, *WA*, d_c*)* $\geq f+1$.

Proof: It is sufficient to show that (SCS_t, SCS_f, WA, d_2) $\geq f+1$. Suppose by contradiction that there is a WA algorithm A in SCS_t and an integer f in $[0, t]$ such that, in every run of A with at most f failures, there are at least two correct processes which decide by round f.

From Lemma 2 it follows that there are two runs of A in SCS_t such that their round f configurations, y and y', satisfy the following: (1) at most f processes have crashed in each configuration, (2) the configurations differ at exactly one process, say p_i, and (3) $val(y) = 0$ and $val(y') = 1$.

From our initial assumption about algorithm A, it follows that (1) in y, there are two alive processes which have decided 0, and (2) in y', there are two alive processes which have decided 1. As y and y' differ at exactly one process and there are two alive processes in y' that have decided 1, it follows that, in y, there is an alive process, say p_j, which has decided 1. Thus, p_j is correct and decides 1 in $r(y)$. However, every correct process in $r(y)$ decides $val(y) = 0$; a contradiction. □

As in the uniform consensus case, Proposition 6 is a generalization of the $f+1$ round global decision lower bound for consensus [20, 1]: in a run with f failures, global decision is $(n-f)$-decision.[7]

Incompatibility. It is easy to design a consensus algorithm that matches either the local decision or the global decision lower bound. In the following proposition, we state that, may be surprisingly, no consensus algorithm can match both the local and the global decision lower bound, even for two consecutive values of f. This is in contrast to uniform consensus where a single algorithm can match both local decision and global decision lower bounds (as we show in Section 7).

Proposition 7 $\forall t \in [1, n-2]$, $\forall f \in [0, t-1]$, *there is no WA algorithm in* SCS_t *that matches the following two conditions: (a) in every run with at most* f *crashes, every correct process decides by round* $f+1$*, and (b) in every run with at most* $f+1$ *crashes, some correct process decides by round* $f+1$.

(Remarks: Condition (a) is for matching the global decision lower bound for f crashes, and condition (b) is for matching the local decision lower bound for $f+1$ crashes. Note that, we do not consider the case $f = t$, because when $f = t$, (a) implies (b), as there is no run in SCS_t with $t+1$ crashes.)

Proof: Suppose by contradiction that there is a WA algorithm A and an integer f in $[0, t-1]$ such that (a) by round $f+1$ of every run with at most f failures, every correct

[7]Notice the case $f = t = n-1$. The $f+1$ round global decision lower bound does not hold when $f = t = n-1$ [20, 1]: in this case, we can design a consensus algorithm that globally decides in $f = t = n-1$ rounds. At first glance, Proposition 6 seems to hold for $f = t = n-1$, and thus imply that global decision requires $f+1$ in this case. However, this observation is flawed because, when $f = t = n-1$, the allowed set of values of c, in Proposition 6, $[2, n-f]$ is $\emptyset$.

process decides, and (b) by round $f+1$ of every run with at most $f+1$ failures, some correct process decides.

From Lemma 2 it follows that, at the end of round f there are two configurations y_0 and y_1 such that (a) at most f processes have crashed in each configuration, (b) the configurations differ at exactly one process, say p_i, and (c) $val(y_0) = 0$ and $val(y_1) = 1$.

Consider run $r(y_0)$. Obviously, $r(y_0)$ is a run with at most f failures, and from our initial assumption, every correct process decides $val(y_0) = 0$ at the end of round $f+1$. Similarly, we construct run $r(y_1)$, which is a failure-free extension of y_1, and every correct process decides $val(y_1) = 1$ at the end of round $f+1$. There are two cases to consider.

Case 1. Process p_i is alive in y_0 and y_1. Consider the extension of y_0 to a run $r'(y_0)$ such that p_i crashes before sending any message in round $f+1$, and no process crashes thereafter. (Recall that $f \leq t-1$.) Notice that $r'(y_0)$ is a run with at most $f+1$ failures and p_i is a faulty process in $r'(y_0)$. Thus, from our initial assumption about A, it follows that there is a correct process $p_j (\neq p_i)$ in $r'(y_0)$ which decides some value $v \in \{0, 1\}$ at round $f+1$. (Notice that, since $p_j \neq p_i$, p_j cannot decide before round $f+1$: as y_0 and y_1 differ only at p_i, if p_j decides by round f, then p_j decides identical values in y_0 and y_1.)

Similarly, consider the extension of y_1 to a run $r'(y_1)$ such that p_i crashes before sending any message in round $f+1$, and no process crashes thereafter. Notice that, at the end of round $f+1$, p_j cannot distinguish $r'(y_1)$ from $r'(y_0)$ because p_j does not receive any message from p_i in round $f+1$ of both runs. Therefore, p_j decides v at the end of round $f+1$ in $r'(y_1)$.

Consider runs $r'(y_{1-v})$ and $r(y_{1-v})$. Since $f \leq n-3$, in run $r'(y_{1-v})$ there is a correct process p_l which is distinct from p_i and p_j. Obviously, p_l is also correct in $r(y_{1-v})$, and hence, p_l decides $val(y_{1-v}) = 1-v$ at the end of round $f+1$ in $r(y_{1-v})$.

Now we construct a run r'' by extending configuration y_{1-v}: process p_i crashes at the beginning of round $f+1$ such that, in round $f+1$, p_l receives a message from p_i but p_j does not receive any message from p_i. No process distinct from p_i crashes in round $f+1$ or a higher round. Obviously, p_j and p_l are correct in r''. At the end of round $f+1$ in run r'', p_j cannot distinguish r'' from $r'(y_{1-v})$ because p_j does not receive a message from p_i in round $f+1$ in both runs. Therefore, p_j decides v at the end of round $f+1$ in r''. However, since p_l receives a message from p_i in round $f+1$, at the end of round $f+1$, p_l cannot distinguish r'' from $r(y_{1-v})$, and therefore, decides $1-v$ at the end of round $f+1$; a contradiction with consensus agreement.

Case 2. Process p_i has crashed in either y_0 or y_1. Without loss of generality, we can assume that p_i has crashed in y_0, and hence, p_i is alive in y_1. (Recall that p_i has different states in the two configurations.) As f processes, including p_i, have crashed in y_0, and p_i has not crashed in y_1, so $f-1$ processes have crashes in y_1. Since $t \leq n-2$ and at most $f-1$ processes have crashed in y_1, there are at least two correct process p_j and p_l (both distinct from p_i) in $r(y_1)$. Consider the run r' which extends y_1 such that process p_i crashes in round $f+1$ and the only alive process that *does not* receive round $f+1$ message from p_i, is p_l, and no process crashes after round $f+1$. Obviously p_j and p_l are correct in r'. At the end of round $f+1$, p_l cannot distinguish $r(y_0)$ from r' because p_l does not receive the round $f+1$

message from p_i in both runs. Thus, p_l decides 0 at the end of round $f+1$ in r'. At the end of round $f+1$, p_j cannot distinguish $r(y_1)$ from r' because both runs extend y_1 and p_j receives round $f+1$ message from p_i in both runs. Thus, p_j decides 1 at the end of round $f+1$ in r'; a contradiction with consensus agreement. $\square$

6.3 Non-Blocking Atomic Commit and Interactive Consistency

Recall that the lower bounds presented in Section 6.1 hold for UC, NBAC, and IC. In the following, we show that for NBAC and IC, the local decision lower bound for the failure-free case ($f=0$) can be shifted to 2. However this result does not hold for UC, as we later exhibit a UC algorithm that locally decides in 1 round in failure-free runs.

Proposition 8 *$\forall t \in [2, n-1]$, (*SCS_t*,* SCS_0*, NBAC, ld) ≥ 2.*

Proof: Suppose by contradiction that there is a NBAC algorithm A such that, in every failure-free run, some process decides in round 1. Let C be the initial configuration in which all processes propose 1. Consider the failure-free run $R1$ starting from C; i.e., $R1 = r(C)$. Suppose that some process p_i decides at the end of round 1. From the abort validity property of NBAC, we know that p_i decides 1 in $R1$.

Consider another run $R2$ such that every process proposes 1. Some process p_j ($\neq p_i$) crashes in round 1 and only p_i receives round 1 message from p_j. Process p_i crashes in round 2, before sending round 2 message to any process, and no process crashes thereafter. At the end of round 1, p_i cannot distinguish $R1$ from $R2$. Thus, p_i decide 1 in $R2$. From NBAC agreement, we know that every process distinct from p_i and p_j decides 1. There exist at least one such process, say p_l, because $t \in [2, n-1]$.

Let $C0$ be the initial configuration in which p_j proposes 0 and all other processes propose 1. Consider a run $R3$ starting from $C0$ with the same failure pattern as $R2$; i.e., p_j crashes in round 1 and only p_i receives round 1 message from p_j, p_i crashes in round 2 before sending round 2 message to any process, and no process crashes thereafter. No process distinct from p_i and p_j can distinguish $R2$ from $R3$: at the end of round 1, only p_i receives message from p_j, but p_i crashes before sending any message in round 2. Therefore, every process distinct from p_i and p_j, decides 1 (as in $R2$), in particular p_l; a contradiction with the commit validity property of NBAC. $\square$

The above proposition highlights a fundamental difference between the time complexity of NBAC and UC in synchronous models. However, the proposition extends to IC. In fact, any IC algorithm can be transformed to a NBAC algorithm (without any additional rounds) as follows. Let $V1$ denote the vector of n components in which every component is 1. Suppose we have an IC algorithm with IC-propose() primitive. We implement the NBAC-propose() primitive of the NBAC specification in the following way. When a process NBAC-proposes $v \in \{0, 1\}$, it IC-proposes v. If a process IC-decides $V1$, then is NBAC-decides 1; if the process IC-decides a vector different from $V1$ then it NBAC-decides 0. Note that the transformation by itself does not require any additional communication, and hence can be performed even in an asynchronous model. Thus, this transformation immediately implies that the bounds in Section 6.1 and Proposition 8 applies to IC.

7 A Matching Synchronous Algorithm

In [23], an NC algorithm was proposed that matches the global decision and global halting lower bounds. The algorithm can be easily modified to derive another algorithm that matches corresponding bounds for UC. However, we knew of no UC algorithm that matches the local decision lower bounds.

In this section, we present an algorithm for IC that *simultaneously* matches the local decision, global decision, and global halting lower bounds for most values of f and t. From our IC algorithm, we then derive matching algorithms for UC and NBAC. (Algorithms that match either the local decision or global decision of NC are straightforward but, as we showed in Proposition 7, no single NC algorithm can match both local and global decision lower bounds.)

7.1 IC algorithm overview

Our IC algorithm (Figure 1) is inspired by the Byzantine Generals algorithm of [23]. Later we show how to modify the algorithm to obtain an NBAC and a UC algorithm.

The algorithm runs for at most $t+1$ rounds. Process p_i maintains two primary variables: (1) an ordered n-tuple est_i, component j of which contains the proposal value of p_j, provided p_i has learnt that value (either directly from p_j or relayed by some other process), and $\perp$ otherwise, and (2) a set of processes $halt_i$ that p_i knows to have either crashed or halted. In each round, the processes exchange their *est* values. If the *halt* set at a process does not change in round k then (1) if the *est* does not change in round k as well, the process decides on its *est* in round k, otherwise, (2) the process decides on its *est* in round $k+1$. Before halting, a process sends a special decision message to all processes, so that the processes can distinguish a halt from a crash.

Roughly speaking, if the *halt* set at a process p_i does not change in a round k, then at the end of round k, no *alive* process has seen more proposal values than p_i. Thus, p_i can decide on est_i at the end of round k, but p_i has to ensure that all other processes also see its *est*. So p_i sends its *est* to all processes in round $k+1$ and then decides. However, if the *est* of p_i does not change in round k, then p_i has already send that *est* to all processes in round k; so p_i can decide at the end of round k.

7.2 Correctness

In the following, a variable *var* at a process p_i is denoted var_i, and if p_i reaches the end of any round r, the value of var_i at the end of round r is denoted var_i^r; var_i^0 denotes the value of the variable at the end of line 5. For $1 \leq r \leq t+1$, $faulty^r$ denotes the set of processes which have crashed by round r, and $faulty^0 = \emptyset$. For any pair of ordered n-tuple d and d', we say that (1) $d = d'$ if for all $j \in [1, n]$, $d[j] = d'[j]$, (2) $d \preceq d'$ if for all $j \in [1, n]$, either $d[j] = \perp$ or $d[j] = d'[j]$, and (3) $d \npreceq d'$ if $d \preceq d'$ is false.

First, we make the following simple observations which we frequently use: (1) (**O1**) for *est* value at any process and any $j \in [1, n]$, $est[j] \in \{v_j, \perp\}$, (2) (**O2**) if, before deciding, p_j receives an EST message from some process p_l in round k, then $newest_l^{k-1} \preceq newest_j^k$. (It

```
at process p_i:
1: procedure propose(v_i)
2:    Ordered n-tuple est_i and newest_i with element i initialized to v_i and all other elements initialized to ⊥
3:    Set halt_i ← newhalt_i ← ∅
4:    Boolean decided_i ← lastRound_i ← false
5:    for 1 ≤ r ≤ t+1 do Multiset S_i^r ← ∅

6:    for round r from 1 to t+1 do
7:       halt_i ← newhalt_i
8:       est_i ← newest_i

9:       {Send phase}
10:      if lastRound_i then
11:         send(r, DEC, est_i) to all
12:      else
13:         send(r, EST, est_i) to all

14:      {Receive phase}
15:      S_i^r ← {est_j | (r, EST, est_j) was received}
16:      if lastRound_i then
17:         if not decided_i then
18:            decide(est_i)                                                    {decision}
19:         return                                                              {halt}
20:      if received any (r, DEC, est_j) then
21:         newest_i ← est_j
22:         lastRound_i ← true
23:      else
24:         newhalt_i ← Π\sender(S_i^r)           {processes from which p_i did not receive any message}
25:         for 1 ≤ j ≤ n do
26:            if there is any est' ∈ S_i^r s.t. est'[j] ≠ ⊥ then newest_i[j] ← est'[j] else newest_i[j] ← ⊥
27:         if newhalt_i = halt_i then
28:            if est_i = newest_i then
29:               decide(est_i); decided_i ← true                               {decision}
30:            lastRound_i ← true
31:      if r = t+1 then
32:         if not decided_i then
33:            decide(newest_i)                                                 {decision}
34:         return                                                              {halt}
```

Figure 1: An early deciding (and halting) interactive consistency algorithm

follows that $newest_j^{k-1} \preceq newest_j^k$.)

Every process decides on some est value; thus, validity immediately follows from Observation O1. Termination follows from the simple observations that no process halts without deciding and no process completes round $t+1$ without halting (lines 31 to 34). In the rest of this section we consider the uniform agreement property. We start with some general lemmas about the algorithm.

Lemma 9 *If no process decides by some round $r \in [1,t]$, then the following holds for any process p_i that completes round r. If $lastRound_i^r = true$, then any process p_j that completes round r, has $newest_j^r \preceq newest_i^r$.*

Proof: We prove the lemma by induction on round number r, such that $1 \leq r \leq t$.

Base case $r=1$. Suppose $lastRound_i^r = true$ and no process decides in round 1. Then p_i has either executed line 22 or line 30 of round 1. Observe that p_i executes line 22 only if some process sends DEC message to p_i. Since *lastRound* is initialized to false and the processes send a DEC messages only when $lastRound = true$, no process has sent a DEC message in round 1. Thus p_i has executed line 30. So $newhalt_i^1 = halt_i^1 = \emptyset$, and hence, $newest_i^1$ contains proposal values of all processes. Thus, any process p_j which completes round 1 has $newest_j^1 \preceq newest_i^1$.

Induction Hypothesis $r = k$: If no process decides by round k then the following holds for any process p_i that completes round k. If $lastRound_i^k = true$, then any process p_j that completes round k, has $newest_j^k \preceq newest_i^k$.

Induction Step $r = k+1 \leq t$. Suppose by contradiction that (1) no process decides by round $r = k+1$, (2) there is a process p_i that completes round $k+1$ such that $lastRound_i^{k+1} = true$ and $newest_i^{k+1} = d'$, and (3) another process p_j completes round $k+1$ with $newest_j^{k+1} = d$ such that $d \not\preceq d'$. Process p_i has either executed line 22 or line 30. If p_i executed line 22, then p_i has received $(k+1,$ DEC$, d')$ message from some process p_l. To send a DEC message in round $k+1$, p_l must have set $lastRound_l$ to $true$ in round k. Thus, from the induction hypothesis, every process that completes round k has $newest^k \preceq d'$. Since $d \not\preceq d'$, process p_j receives a round $k+1$ message from some process with a n-tuple d'' such that $d'' \not\preceq d'$; a contradiction because, for all processes which complete round k, $newest^k \preceq d'$. Hence, p_i executed line 30, and $halt_i^{k+1} = newhalt_i^{k+1}$. Since p_j completes round $k+1$, p_i received the round $k+1$ message from p_j containing $newest_j^k$, and hence, $newest_j^k \preceq newest_i^{k+1} = d'$. As $newest_j^{k+1} = d \not\preceq d'$, it follows that p_j received $(k+1, *, d'')$ from some process p_m such that $d'' \not\preceq d'$, and p_i did not receive $(k+1, *, d'')$ from p_m (otherwise, $d'' \preceq newest_i^{k+1} = d'$). Thus $p_m \in newhalt_i^{k+1}$. However, as p_m completed round k, $p_m \notin newhalt_i^k = halt_i^{k+1}$. Thus, $halt_i^{k+1} \neq newhalt_i^{k+1}$; a contradiction. □

Lemma 10 *If a process p_i has $halt_i^r = newhalt_i^r$ in some round $r \in [1,t]$, then either p_i halts by round $r+1$ or crashes by round $r+1$.*

Proof: Obvious from the algorithm. □

Lemma 11 *If no process halts by some round $r-1 \in [1,t]$, and if there is a process p_i such that, for every round number $r' \in [1,r]$, $halt_i^{r'} \neq newhalt_i^{r'}$, then $|faulty^r| \geq r$.*

Proof: Suppose there is a round r such that no process halts by round $r-1$ and there exists a process p_i such that, for every round number $r' \in [1,r]$, $halt_i^{r'} \neq newhalt_i^{r'}$. Clearly, $halt_i^{r'} = newhalt_i^{r'-1} \subsetneq newhalt_i^{r'}$. Thus $|newhalt_i^r| \geq r$. Any process in $newhalt_i^r$ has either halted by round $r-1$ or crashed by round r. Since no process halts by round $r-1$, $newhalt_i^r \subseteq faulty^r$, and hence, $|faulty^r| \geq r$. □

Lemma 12 *If no process halts by round $r+1 \in [1,t]$, then $|faulty^r| \geq r$.*

Proof: The proof is trivial for $r+1 = 1$. So we consider the case $r+1 \in [2,t]$. Consider any correct process p_i. Since p_i does not halt by round $r+1$, it follows from Lemma 10 that for

$r' \in [1, r]$, $halt_i^{r'} \neq newhalt_i^{r'}$. Since no process halts by round $r-1$, applying Lemma 11, we have $|faulty^r| \geq r$. $\square$

Lemma 13 *If every process which decides, decides in line 29 of round $t+1$ or line 33 of round $t+1$, then $|faulty^t| \geq t$.*

Proof: Consider any correct process p_i. Since p_i does not decide in line 18 of round $t+1$, $lastRound_i^t = false$. Thus $newhalt_i^t \neq halt_i^t$ (from lines 27 and 30). Furthermore, as p_i does not halt by round t, for every $r \in [1, t-1]$, $newhalt_i^r \neq halt_i^r$. Thus for every $r \in [1, t]$, $newhalt_i^r \neq halt_i^r$. Observing that no process halts by round $t-1$ and applying Lemma 11, we have $|faulty^t| \geq t$. $\square$

Lemma 14 *(Uniform Agreement) No two processes decide differently.*

Proof: If no process decides then the lemma trivially holds. Suppose some process decides. Consider the lowest round number r in which some process decides, and consider any process p_i that decides in round r, say on some n-tuple d. We divide the proof into two parts: (a) p_i does not decide in line 33 of round $t+1$, and (b) $r = t+1$ and p_i decides in line 33 of round $t+1$.

(a) *p_i does not decide in line 33 or round $t+1$:* Process p_i decides either in (1) line 18 or (2) line 29 of round r. In both cases, we show the following: no process can decide a n-tuple different from d in round r, and any process that completes round r without deciding, does so with $newest^r = d$. This immediately implies uniform agreement. (Note that, even if $r = t+1$, and another process p_j decides in line 33 of round r, p_j decides on $newest_j^r = d$.)

Process p_i decides in line 18 of round r: Notice that $r > 1$ because no process can decide at line 18 in round 1 (as $lastRound^0 = false$). Since p_i decides in line 18, $lastRound_i^{r-1} = true$ and p_i sends a DEC message in round r. We claim that every DEC message sent in round r is (r, DEC, d). Suppose that another process p_j sends a (r, DEC, $d1$) message. Then $lastRound_j^{r-1} = true$. Since no process decides by round $r-1$, applying Lemma 9 twice we have $d1 = newest_j^{r-1} \preceq newest_i^{r-1} = d$ and $d = newest_i^{r-1} \preceq newest_j^{r-1} = d1$, i.e, $d1 = d$. As p_i completes round r, every process receives at least one (r, DEC, d) message, and either decides d in line 18, or adopts d as *newest* in line 21.

Process p_i decides in line 29 of round r: We claim that no process decides a value different from d in round r. Clearly, p_i does not receive any DEC message in round r (otherwise, p_i does not execute line 29). Suppose some process p_j decides $d1$ in round r. If process p_j decides in line 18, then p_j sends DEC message in round r, and p_i receives that message (because p_j completes round r); a contradiction. Thus p_j decides in line 29. From the predicate at line 28, if follows that p_i sent (r, EST, d) in round r and p_j sent (r, EST, $d1$) in round r. Since p_i receives round r message from p_j and vice versa, $d1 \preceq d$ and $d \preceq d1$, i.e., $d = d1$.

We now show that any process that completes round r without deciding at line 18 or line 29, does so with $newest = d$. Suppose by contradiction that some process p_j completes round r with $newest = d2 \neq d$ and without deciding in line 18 or 29. Process p_j updates its variable *newest* in line 21 or line 26. Suppose p_j updated its *newest* in line 21. Then p_j has received a DEC message from some process p_m. Since p_i decides at line 29, it does not receive any DEC message in round r. Thus $p_m \in newhalt_i^r$. Since p_m completes round $r-1$, $p_m \notin newhalt_i^{r-1} = halt_i^r$ (if $r = 1$ then obviously $p_m \notin halt_i^r = \emptyset$.). Hence, the predicate at line 27 evaluates to false at p_i, and p_i cannot decide in line 29; a contradiction. Thus, p_j updates its *newest* in line 26. Since p_i completes round r by deciding d and evaluates the condition in line 28 to true, p_i sends a (r, EST, d) in round r. Thus p_j receives (r, EST, d) from p_i, and hence, $d \preceq d2$. As $d2 \neq d$, it follows that $d2 \npreceq d$. Consequently, there is a process p_m such that p_j receives $d3 \npreceq d$ from p_m, and p_i does not receive any message from p_m in round r. Thus, $p_m \in newhalt_i^r$. However, p_m completes round $r-1$ and hence, $p_m \notin newhalt_i^{r-1} = halt_i^r$ (if $r = 1$ then obviously $p_m \notin halt_i^r = \emptyset$). Hence, the predicate at line 27 evaluates to false at p_i, and p_i cannot decide in line 29; a contradiction.

(b) $r = t+1$ *and* p_i *decides in line 33 of round* $r = t+1$*:* From the definition of r, every process which decides, decides in round $t+1$. We have shown above that, if any process decides in line 18 or line 29 of round $t+1$, then every process which decides in round $t+1$, decides the same value. Therefore, we need to only consider the case where every process which decides, decides at line 33 of round $t+1$. From Lemma 13, we have $|faulty^t| \geq t$. Since at most t processes may crash in a run, $|faulty^t| = t$, and hence, every process which enters round $t+1$, is a correct process. Consequently, every process which enters round $t+1$, receives the same set of messages in round $t+1$. Observe that no process sends DEC message in round $t+1$ (otherwise, that process decides in line 18 of round $t+1$ or line 29 of round t; a contradiction). Thus every process which enters round $t+1$, updates *newest* to the same value in line 26, and decides on identical values in line 33. □

7.3 Time-complexity

We now discuss the time complexity properties of our IC algorithm. In the following lemma we show that, in runs with $f \geq 1$ failures, the algorithm achieves local decision in $f+1$ rounds and global decision in $f+2$ rounds. However, when $f = 0$, the local decision takes the same number of rounds as global decision (2 rounds) - recall that, we showed in Proposition 8 that NBAC (and hence, IC) algorithms require 2 rounds for local decision when $f = 0$. (In Section 7.4, we show a UC algorithm that achieves local decision in 1 round when $f = 0$.)

We say that a process p_i *learns* index $l \in [1,n]\backslash\{i\}$ in round k if $newest_i^{k-1}[l] = \perp$ and $newest_i^k[l] \neq \perp$. (In other words, p_i learns about the proposal value of p_l in round k.) We say that p_i learns index i in round 0. We say that p_i learns index l from p_j in round k if $newest_i^{k-1}[l] = \perp$ and p_i receives a round k message from p_j containing an *est* such that $est[l] \neq \perp$; we also say that p_j propagates index l in round k. (Note that there may be more than one process from which a process learns the same index index in a round.) Obviously, if p_i propagates l in round k, then p_i learns l in a lower round.

Lemma 15 *In any run with at most f faulty processes, the following properties hold:*
(a) if $f \in [1,t]$, then there is a correct process that decides by round $f+1$.

(b) if $f \in [0, t-2]$, then any process that halts, halts by round $f+2$.
(c) Any process that halts, halts by round $t+1$.

Proof: (a) For $f = t$, the proof is trivial because every correct process decides by round $t+1$. Consider a run in which at most $f \in [1, t-1]$ processes crash, and suppose, by contradiction that no correct process decides by round $f+1$. Thus, no process has halted by round $f+1 \leq t$. It follows from Lemma 12 that $|faulty^f| \geq f$. Since at most f processes fail in the run, $|faulty^f| = f$ and every process which enter round $f+1$ is correct. Furthermore, since no correct process halts by round f, from Lemma 12 $|faulty^{f-1}| \geq f-1$. Since $|faulty^f| = f$, at most one process crashes in round f.

Let S be the set of processes which enter round $f+1$. Since every process in S is correct, all of them complete round $f+1$. We establish a contradiction by showing that some process in S executes line 29 of round $f+1$. We demonstrate this fact indirectly by showing the following four claims for processes in S in round $f+1$: (1) every process has $lastRound = false$ in line 16, (2) no process receives a DEC message in round $f+1$, (3) every process evaluates the predicate in line 27 to true, and (4) some process evaluates the predicate in line 28 to true.

Claim 1. If $lastRound = true$ at a correct process in line 16 of round $f+1$ then the process halts in round $f+1$. As we know that every process in S is correct, and no process has halted by round $f+1$, Claim 1 follows immediately.

Claim 2. Suppose by contradiction that some process $p_i \in S$ receives a DEC message from some process p_j in round $f+1$. Since every process which enters round $f+1$ is correct, p_j is a correct process, and hence, p_j decides in line 18 of round $f+1$ or line 29 of round f; a contradiction. Thus no process in S receives a DEC message in round $f+1$.

Claim 3. Suppose by contradiction that some process $p_i \in S$ evaluates the predicate at line 27 to false; i.e., $halt_i^{f+1} \neq newhalt_i^{f+1}$. Since p_i does not halt by round $f+1$, from Lemma 10 we have, $halt_i^k \neq newhalt_i^k$ for every k in $[1, f]$. Thus $halt_i^k \neq newhalt_i^k$ for every k in $[1, f+1]$. As no correct process halts before round $f+2$, from Lemma 11 it follows that $|faulty^{f+1}| \geq f+1$; a contradiction.

Claim 4. Suppose by contradiction that every process in S evaluates the predicate in line 28 to false. It follows that, in round $f+1$, every process in S learns an index. (Recall that every process that enters round $f+1$ is correct and is in set S.)

Consider any process $p_i \in S$ which learns index $l1$ in round $f+1$ from some process p_x. Suppose p_x learns index $l2$ in round $f+1$ from process p_y. Since p_i learns from p_x and p_x learns from p_y, $p_i \neq p_x$ and $p_x \neq p_y$. (Note that p_i and p_y may not be distinct.) Since p_x propagates $l1$ and learns $l2$, $l1 \neq l2$.

Since p_x is a correct process, p_x learns $l1$ in round f (otherwise, if p_x learned $l1$ in a round lower than f, p_x would have propagated $l1$ to p_i by round f). Similarly, p_y learns $l2$ in round f. Consider the process p'_x from which p_x learns $l1$ in round f. Process p'_x must have crashed in round f, otherwise, on receiving the round f message from p'_x, p_i would have learned $l1$ in round f. Similarly, the process p'_y from which p_y learns $l2$ in round f must have crashed in round f, otherwise, p_x would have learned $l2$ from p'_y in round f. We claim that p'_x and p'_y are distinct processes. Otherwise, if $p'_x = p'_y$, then p'_x propagates both $l1$ and $l2$ in round f, and when p_x receives a message from p'_x in round f, p_x learns both $l1$ and $l2$ in round f; a contradiction. (Recall that we assumed p_x learned $l2$ in round $f+1$.)

Thus two processes, p'_x and p'_y, crashes in round f. However, recall that we have already shown that at most one process crashes in round f; a contradiction.

(b) Consider a run with at most $f \in [0, t-2]$ processes crash, and suppose by contradiction that a process p_i completes round $f+2$ without halting. Observe that, if any process p_j halts at round $k \leq f+1$ then p_j sends a DEC message in round k. Since p_j completes round k, p_i receives the DEC message, sets $lastRound$ to $true$ in round k, and halts in round $k+1 \leq f+2$. Thus no process halts by $f+1$. Since p_i does not halt by round $f+2$, for every $r \in [1, f+1]$, $newhalt_i^r \neq halt_i^r$. Applying Lemma 11 we have $|faulty^{f+1}| \geq f+1$; a contradiction.

(c) Obvious from the algorithm. □

7.4 Deriving NBAC and UC algorithms

In Section 6.3, we showed how to transform any IC algorithm to an NBAC algorithm, without any additional communication. An equally straightforward transformation generates a UC algorithm from an IC algorithm: on UC-propose(v), a process invokes IC-propose(v), and if a process IC-decides an n-tuple d, then it UC-decides $d[l]$ where l is the lowest index such that $d[l] \neq \perp$.

The IC algorithm of Figure 1 does not locally decide in round 1 in a failure-free run ($f = 0$). Therefore, to match the local decision lower bound for UC when $f = 0$, we modify the UC algorithm obtained from our IC algorithm, by adding the following: p_1 UC-decides on its proposal value v_1 in the receive phase of round 1. This modification does not violate UC agreement because, if p_1 completes the send phase of round 1, then every process which completes round 1 has $est[1] = v_1$. Subsequently, at all processes, $est[1]$ is always v_1. Thus no process can UC-decide a value different from v_1.

7.5 Synchronous results summary

Combining our lower bound results with the time-complexity of the IC algorithm, the derived NBAC and UC algorithms, and the simple NC algorithm sketched in the introduction, we get the following tight bounds:

1. $\forall t \in [1, n-1]$, $\forall f \in [0, t-1]$, $(SCS_t,\ SCS_f,\ \text{UC},\ ld) = f+1$.
   Local decision bound for uniform consensus.

2. (a) $\forall t \in [1, n-1]$, $\forall f \in [1, t-1]$, $\forall P \in \{\text{NBAC, IC}\}$, $(SCS_t,\ SCS_f,\ \text{P},\ ld) = f+1$.
   (b) $\forall t \in [1, n-1]$, $\forall P \in \{\text{NBAC, IC}\}$, $(SCS_t,\ SCS_0,\ \text{P},\ ld) = 2$.
   Local decision bounds for non-blocking atomic commit and interactive consistency.

3. $\forall t \in [3, n-1]$, $\forall f \in [0, t-3]$, $\forall c \in [2, n-f]$, $\forall P \in \{$UC, NBAC, IC$\}$, (SCS_t, SCS_f, P, d_c) $= f+2$. For $c \geq 2$, c-decision bounds for uniform consensus, non-blocking atomic commit and interactive consistency. (Recall that 1-decision is local decision.)

4. $\forall t \in [1, n-1]$, $\forall f \in [0, t]$, (SCS_t, SCS_f, NC, ld) $= f$.
Local decision bound for consensus.

8 Eventually Synchronous Lower Bound

In this section we investigate lower bounds for UC in eventually synchronous model ES_t. We do not consider lower bounds for NBAC and IC in ES_t because they are impossible to solve in ES_t if $t \geq 1$. Furthermore, any algorithm that solves consensus also solves uniform consensus in ES_t [15]. Thus, in ES_t, we only investigate lower bounds for uniform consensus.

We know from [14] that every UC algorithm in ES_t has a run which requires an arbitrary number of rounds for any correct process to decide (because a run may remain "asynchronous" for an arbitrary number of rounds). Thus, we focus on *synchronous* runs of ES_t: runs in which $GSR = 1$. (In other words, a run of ES_t is synchronous if it is also a run of SCS_t.)

As all runs of SCS_t are synchronous runs of ES_t, the local and global decision lower bounds for UC in SCS_t, also holds for synchronous runs of ES_t; i.e., roughly speaking, local decision lower bound is $f+1$ and global decision lower bound is $f+2$. However, we knew of no algorithm that showed that the bounds are tight, except when $f = 0$ and $f = t$ (the best and the worst case): the global decision tight bound is 2 rounds in runs with $f = 0$ crashes [19, 31, 25], and $t+2$ rounds in runs with $f = t$ crashes [6].

In following proposition, we show that, for most values of f, the local decision lower bound is $f+2$ rounds, same as the lower bound for global decision. (We give a matching algorithm in Section 9.) We exhibit the lower bound for UA; the bound immediately extends to UC because any UC algorithm also solves UA. The proposition says that, every UA algorithm in ES_t has a run in SCS_f (i.e., a synchronous run with f crashes) in which every correct process decides in round $f+2$ or a higher round.

Proposition 16 *$\forall t \in [1, n-1]$, $\forall f \in [0, t-3]$, (*ES_t*,* SCS_f*, UA, ld)* $\geq f+2$.

Proof: Suppose by contradiction that there is a UA algorithm A in ES_t and an integer f in $[0, t-3]$ such that, in every synchronous run of A with f crashes some correct process decides by round $f+1$. We observe that any UA algorithm also solves WA, and every run of an algorithm in $SCS1_t$ is also a run in SCS_t. Thus, Lemma 2 holds when WA and $SCS1_t$ are replaced with UA and SCS_t, respectively. So there are two runs of A in SCS_t such that their round f configurations, y and y', satisfy the following: (1) at most f processes have crashed in each configuration, (2) the configurations differ at exactly one process, say p_i, and (3) $val(y) = 0$ and $val(y') = 1$. (Recall that, given a configuration C, $r(C)$ and $val(C)$ are defined only if C is a configuration of a run in a synchronous model.)

Let z and z' denote the configurations at the end of round $f+1$ of $r(y)$ and $r(y')$, respectively. As at most f processes crash in each run, $r(y)$ and $r(y')$, so it follows from our assumption about algorithm A that, some correct process decides by round $f+1$ in each run. Thus, there is at least one alive process in z, say q_1, that has decided 0. Similarly, there is at least one alive process in z', say q_3, that has decided 1. There are three cases to consider.

Case 1. $p_i \notin \{q_1, q_3\}$. This case is exactly similar to the case in the proof of Proposition 4. We can derive a contradiction by constructing the same runs R1, R2, R12, and R21 in SCS_t. (These are valid runs of A in ES_t because SCS_t is a submodel of ES_t.)

Case 2. $p_i \in \{q_1, q_3\}$ and p_i is alive in both y and y'.[8] Without loss of generality we can assume that $p_i = q_1$. (Note that the proof holds even if $p_i = q_1 = q_3$.) Consider the following three runs (R3 is a synchronous run, whereas R4 and R5 are non-synchronous runs.):
R3 is a run such that (1) the round f configuration is y, (2) p_i crashes in round $f+1$ before sending any message, (3) if $q_3 \neq p_i$ then q_3 crashes before sending any message in round $f+2$, and every message sent by q_3 in round $f+1$ is received in round $f+1$, (4) no process distinct from p_i and q_3 crashes after round f, and (5) no message is delayed. Since $t \leq n-1$, there is at least one correct process in $R3$, say p_l. Suppose p_l decides $v \in \{0,1\}$ in some round $K1 \geq f+1$. (To see why p_l cannot decide before round $f+1$ in $R3$, notice that the state of p_l at the end of round f is the same in runs $r(y)$, $r(y')$ and $R3$, because $p_l \neq p_i$. If p_l decides v before round $f+1$ in $R3$ then it also decides v in $r(y)$ and $r(y')$. However, $val(y) \neq val(y')$.)
R4 is a run such that (1) the round f configuration is y, (2) p_i and q_3 crash before sending any message in round $f+2$, and only p_i and q_3 receive the round $f+1$ message from p_i (all other round $f+1$ messages from p_i are lost[9]), (3) if $q_3 \neq p_i$, every process which completes round $f+1$ receives round $f+1$ message from q_3, (4) no process distinct from p_i and q_3 crashes after round f, and (5) no message is delayed. Notice that p_i cannot distinguish the configuration at the end of round $f+1$ in $R4$ from z, and thus, p_i decides 0 at the end of round $f+1$ in $R4$ (because $p_i = q_1$). However, p_l cannot distinguish round $K1$ configuration of $R4$ from that of $R3$ because (a) at the end of round f, the two runs are different only at p_i, and all round $f+1$ messages sent by p_i to processes distinct from p_i and q_3 are lost, and (b) p_i and q_3 do not send messages after round $f+1$. Thus (as in $R3$) p_l decides v in round $K1$.
R5 is a run such that[10] (1) the round f configuration is y', (2) p_i and q_3 crash before sending any message in round $f+2$, and only p_i and q_3 receive the round $f+1$ message from p_i (all other round $f+1$ messages from p_i are lost), (3) if $q_3 \neq p_i$, then every process which completes round $f+1$ receives round $f+1$ message from q_3, (3) no process distinct from p_i and q_3 crashes after round f, and (5) no message is delayed. Notice that q_3 cannot distinguish the configuration at the end of round $f+1$ in $R5$ from z' (because in both runs, q_3 receives the message from p_i in round $f+1$), and thus, q_3 decides 1 at the end of round $f+1$ in $R5$.

[8] To see why we cannot reuse the proof of Proposition 4, observe that, if $p_i = q_1$ then run $R1$ (in the proof of Proposition 4) is not a run of A in SCS_t: in SCS_t, p_i cannot crash in the send phase of round $f+1$, and then decide at the end of round $f+1$. Similarly, if $p_i = q_3$ then run $R2$ is not a run in SCS_t. Hence we construct runs of A in ES_t which are not in SCS_t (i.e., non-synchronous runs), to derive a contradiction.

[9] From the definition of ES_t, messages sent by a faulty process (p_i) may be lost in a non-synchronous run.

[10] R5 extends y' in the same way as R4 extends y.

However, p_l cannot distinguish round $K1$ configuration of $R5$ from that of $R3$ because, (a) at the end of round f the two runs are different only at p_i, and all round $f+1$ messages sent by p_i to processes distinct from p_i and q_3 are lost, and (b) p_i and q_3 do not send messages after round $f+1$. Thus (as in $R3$) p_l decides v in round $K1$.

Clearly, either $R4$ or $R5$ violates uniform agreement: p_l decides v in both runs, however, p_i decides 0 in $R4$ and q_3 decides 1 in $R5$.

Case 3. $p_i \in \{q_1, q_3\}$ and p_i has crashed in either y or y'. (Process p_i has not crashed in both y and y' because p_i has different states in y and y'.) Notice that the case $p_i = q_1 = q_3$ is not possible because, in that case, p_i is alive in both z and z', and hence in y and y'. We show the contradiction for the case when $p_i = q_1 \neq q_3$. (The contradiction for $p_i = q_3 \neq q_1$ is symmetric.)

Since, $p_i = q_1$, p_i is alive in z, and hence, alive in y. Thus p_i has crashed in y'. Consider the following non-synchronous run:
R6 is a run such that (1) the round f configuration is y, (2) in round $f+1$, only p_i receives the round $f+1$ message from itself (all other messages sent by p_i in round $f+1$ are lost), (3) p_i crashes in round $f+2$ before sending any message in that round, (4) no process distinct from p_i crashes after round f, and (5) no message is delayed. At the end of round $f+1$ in $R6$, p_i cannot distinguish the configuration from z, and therefore, decides 0 (because $p_i = q_1$). However, q_3 does not receive the round $f+1$ message from p_i in $R6$, and hence, q_3 cannot distinguish the configuration at the end of round $f+1$ in $R6$ from z'. (Observe that, in z', q_3 does not receive the round $f+1$ message from p_i because p_i has crashed in y'.) Consequently, q_3 decides 1 in $R6$; a contradiction with uniform agreement. □

Remark. A closer look at the proof of Proposition 16 reveals that the non-synchronous runs we construct (R4, R5, and R6) require only a small amount of non-synchrony in the model. The three runs are valid in a model if the model allows the following: if some message from process p_i is lost in round $f+1$, then p_i might complete round $f+1$. (Recall that, in a synchronous model, if some messages from p_i is lost in round $f+1$ then p_i has necessarily crashed in send phase of round $f+1$.) It is easy to see that such runs can also be constructed in the synchronous send-omission model [17] as well as in an asynchronous round-by-round model enriched with a *Perfect* failure detector [5]. Thus the $f+2$ local decision lower bound in synchronous runs also extend to these two models.

9 A Matching Eventually Synchronous Algorithm

In this section, we present a UC algorithm in ES_t that matches the local and global decision lower bounds in synchronous runs. We assume that $t < n/2$, as UC is impossible to solve in ES_t if $t \geq n/2$ [8]. As we pointed out earlier, [19, 31, 25] give a UC algorithm in ES_t that matches the global decision bound for synchronous runs with $f = 0$ crashes, and [6] gives a UC algorithm in ES_t that matches the global decision bound for synchronous runs with $f = t$ crashes. We knew of no UC algorithm that matches the bounds for $1 \leq f \leq t-1$.

Figure 2 presents a uniform consensus algorithm A_{es} in ES_t that globally decides (and

```
at process p_i:
procedure propose(v_i)
  est_i ← v_i
  for round s_i from 1 to ∞ do
    k_i ← ((s_i − 1) mod (t + 2)) + 1                              {k_i varies from 1 to t + 2}
    if k_i = 1 and STATE_i ≠ DECIDE then
      Halt_i ← ∅
      STATE_i ← SYNC1                    {STATE_i is either SYNC1, SYNC2, NSYNC, or DECIDE}

    {Send phase}
    send(s_i, est_i, STATE_i, Halt_i) to all

    {Receive phase}
    wait until received messages in round s_i
    if STATE_i = DECIDE then
      return                                                                      {halt}
    if received any (s', est', DECIDE, ∗) with s' ≤ s_i then
      est_i ← est'; decide(est_i); STATE_i ← DECIDE; go to the next round        {decision}
    if STATE_i ∈ {SYNC1, SYNC2} then
      Halt_i ← Halt_i ∪ {p_j | (p_i received(s_i, ∗, NSYNC, ∗) from p_j) or
      (p_i received(s_i, ∗, ∗, Halt_j) from p_j s.t. p_i ∈ Halt_j) or (p_i did not receive any round s_i message from p_j)}
      msgSet_i ← { m | m is a round s_i message received from p_j ∉ Halt_i}
      est_i ← Min{est | (∗, est, ∗, ∗) ∈ msgSet_i}
      if (STATE_i = SYNC2) and (|Halt_i| ≤ t) and (STATE = SYNC2 for every message in msgSet_i) then
        decide(est_i); STATE_i ← DECIDE; go to the next round                    {decision}
      if |Halt_i| ≤ k_i − 1 then
        STATE_i ← SYNC2
      if k_i ≤ |Halt_i| ≤ t then
        STATE_i ← SYNC1
      if |Halt_i| > t then
        STATE_i ← NSYNC
    if (STATE = NSYNC) and (received any (s_i, est', SYNC2, ∗)) then
      est_i ← est'
```

Figure 2: A uniform consensus algorithm A_{es} in ES_t

hence, locally decides) within $f + 2$ rounds in every synchronous run with f crashes, for $0 \le f \le t$. In other words, our algorithm matches the $f+2$ round global (and local) decision lower bound for synchronous runs of UC algorithms in ES_t.

9.1 Overview

Algorithm A_{es} is a generalization of the UC algorithm of [6] modified for early decision. A_{es} assumes the following: (1) the model ES_t with $t \in [0, (n-1)/2]$ (i.e., a majority of processes are correct), (2) any message sent by a process p_i to itself in any round k, is either received in round k, or p_i crashes in round k, and (3) the set of proposal values in a run is a totally ordered set, e.g., every process p_i can tag its proposal value with its index i and then the values can be ordered based on this tag.

The algorithm A_{es} proceeds in sessions, where each session is composed of $t+2$ rounds of message exchange. A run globally decides within $f + 2$ rounds in a “synchronous” session, provided at most f processes crash in the run. In each round of a session, processes exchange their estimate (of the decision value), and roughly speaking, adopt the minimum estimate value seen in the round as the estimate for the next round. In this respect, a session of A_{es} is similar to the UC algorithm presented in Section 7.4: if the model was synchronous, then a process p_i could simply monitor the set of processes from which p_i did not receive any messages (set $Halt_i$), and then, p_i could decide on its own estimate when $Halt_i$ did not change for a round. Basically, p_i could do so because, in synchronous models, $Halt_i$ would be equal to the set of crashed processes, and hence, if $Halt_i$ did not change for a round, then p_i would have the smallest estimate among all alive processes.

However, in ES_t, even if p_i does not receive a message from some process p_j, p_j might not have crashed, and could continue sending messages in subsequent rounds. Thus, even if $Halt_i$ does not change for a round, p_i might not have the lowest estimate among all alive processes. Therefore, in A_{es}, in addition to the estimate value, processes also exchange the $Halt$ sets to detect whether the current session is synchronous. Furthermore, to ensure early decision, p_i maintains and exchanges the variable STATE$_i$ which says if p_i considers the session to be synchronous (SYNC1), or if p_i can decide in the next round (SYNC2), or whether p_i considers the session to be asynchronous (NSYNC).

9.2 Description

The processes are supposed to invoke propose($*$) with their respective proposal values as a parameter, and the procedure progresses in *sessions*: a session consists of $t+2$ rounds, and session sn contains rounds from $((sn-1)*(t+2))+1$ to $sn*(t+2)$. We call the k^{th} round in a session sn (i.e., round $((sn-1)*(t+2))+k$) as *step* k of session sn. Recall that, for every run R in ES_t, there is a unknown round number GSR from which the system is *synchronous* (property 4 of ES_t). We say that a session is synchronous if the session starts in round GSR or in a higher round.

Every process p_i maintains the following variables:

k_i is the current round number;

STATE$_i$ at p_i reflects its view on how much progress is made towards achieving a decision in the current session - (1) if STATE$_i$ is updated to NSYNC then p_i considers the current session to be asynchronous, (2) if STATE$_i$ is updated to SYNC1 then p_i considers the session to be synchronous but p_i cannot decide at the next round, (3) if STATE$_i$ is updated to SYNC2 then p_i considers the session to be synchronous with the possibility of a decision in the next round, (4) and upon decision, p_i updates STATE$_i$ to DECIDE;

est_i is the estimate of the possible decision value, and roughly speaking, the minimum value seen by p_i;

$Halt_i$ is a set of processes p_j such that, in the current round or a lower round of this session, at least one of the following occurred: p_i received STATE = NSYNC from p_j, p_i did not receive a message from p_j, or p_j did not receive a message from p_i;

$msgSet_i$ is a set of messages received by p_i from processes that are not in $Halt_i$.

The variables are initialized as follows. Round number k_i starts from 1 and est_i value is initialized to the proposal value of p_i. Variables STATE$_i$ and $Halt_i$ are initialized to SYNC1 and $\emptyset$ respectively, and are reset to their initial values at the beginning of each session. In each round, processes exchange STATE, est, and $Halt$ variables, update their own variable depending upon the messages received, and possibly decide. In round k, p_i updates its variables as follows.

1. If p_i receives a DECIDE message, then p_i decides on the decision value received.

2. If STATE$_i$ is SYNC1 or SYNC2 then

   - p_i updates $Halt_i$ to include all processes already in $Halt_i$, and the set of processes p_j such that: (a) p_i has received an NSYNC message from p_j in round k, (b) received a message from p_j with $p_i \in Halt_j$ in round k, or (3) p_i has not received any message from p_j in round k.

   - p_i includes in $msgSet_i$ every message received in round k whose sender is not in $Halt_i$, and p_i computes est_i to be the minimum est value among messages in $msgSet_i$.

   - if STATE$_i$ is SYNC2, $Halt_i$ is of at most size t, and all messages in $msgSet_i$ contains STATE = SYNC2, then p_i decides on its estimate.

   - Depending on the size h of set $Halt_i$, p_i updates STATE$_i$ as follows: if h is lower than the current round number, then STATE$_i$ is set to SYNC1, else if h is at most t then STATE$_i$ is set to SYNC2, otherwise, STATE$_i$ is set to NSYNC.

3. If STATE = NSYNC and p_i receives any message with STATE = SYNC2, then p_i adopts the estimate contained in the message.

4. Upon decision in round k, p_i sends the decision value to all processes in round $k+1$, and then halts.

9.3 Correctness

The validity property of the algorithm follows from the following three simple observations: (1) the est value of a process is initialized to the proposal value of the process, (2) est value of a process at the beginning of round $s \geq 2$ is the est value of some process at the beginning of round $s-1$, and (3) every process decides on the est value of some process. In the rest of the section, we prove the uniform agreement property of the algorithm. We defer the proof of termination property to the next subsection, where we prove termination along with the time-complexity property of the algorithm.

For a given session, we introduce the following notations. For any variable val_i at process p_i, we denote by $val_i[k]$ ($k \geq 1$) the value of the variable val_i immediately after the completion of step k; $val_i[0]$ denotes the value of val_i immediately before sending messages in step 1. We assume that there is a symbol *undefined* which is distinct from any possible value of the variables in the algorithm. If p_i crashes before completing step k, then $val_i[k] =$ *undefined*; if p_i crashes before sending messages in step 1, then $val_i[0] =$ *undefined*. We first prove the following lemma.

Lemma 17 *Consider a process p_l, a session sn, and a step k in that session, such that* $\text{STATE}_l[k] \in \{\text{SYNC}1, \text{SYNC}2\}$ *(i.e., p_l completes step k with* $\text{STATE} = \text{SYNC}1$ *or* $\text{STATE} = \text{SYNC}2$*). Let $senderMS_l[k]$ be the set of processes that have sent the messages that are in $msgSet_l[k]$. Then, $senderMS_l[k] = \Pi - Halt_l[k]$.*

Proof: Process p_l completes step k with $\text{STATE} = \text{SYNC1}$ or $\text{STATE} = \text{SYNC2}$, and hence, updates $Halt$ and $msgSet$ at line 17 and line 18 of step k, respectively. Consider any process $p_m \in \Pi$. There are two exhaustive and mutually exclusive cases regarding the message from p_m to p_l in step k:

- If p_l does not receive the messages from p_m in step k, then from the third condition in line 17, $p_m \in Halt_l[k]$, and from line 18, $p_m \notin senderMS_l[k]$.

- If p_l receives the step k message from p_m, then from line 18, $p_m \in senderMS_l[k]$ if and only if $p_m \notin Halt_l[k]$. □

Lemma 18 *(Uniform agreement) No two processes decide differently.*

Proof: If no process ever decides then the lemma is trivially true. Thus, consider the lowest session sn in which some process decides. In session sn, consider the lowest step in which some process decides, say step $k' + 1 \geq 2$. (It is easy to see that no process can decide in step 1 of sn.) If some process decides in line 15, then some other process has decided in a lower step of sn or in a lower session; a contradiction with the definition of $k' + 1$ and sn. Thus some process decides in line 21 of step $k' + 1$. We claim the following:

Claim 19. *(Elimination) If there are two processes p_x and p_y such that* $\text{STATE}_x[k'] \in \{\text{SYNC}1,\text{SYNC}2\}$ *and* $\text{STATE}_y[k'] = \text{SYNC}2$ *then $est_x[k'] \geq est_y[k']$.*

[**Proof of Lemma 18 continued**] For now, we assume the claim, and prove uniform agreement. We later give the proof of Claim 19. Suppose that some process p_w decides d at line 21 of step $k'+1$. From line 20 it follows that, there is a message in $msgSet_w[k'+1]$ that has $\text{STATE} = \text{SYNC2}$ and $est = d$, say from process p_v. Consider another process p_u which completes step k' with $\text{STATE} = \text{SYNC2}$ and $est = d'$. In Claim 19, if we substitute p_x by p_v and p_y by p_u, then $d \geq d'$. Similarly, if we substitute p_x by p_u and p_y by p_v, then $d' \geq d$. Thus $d' = d$, and it follows that, any process which completes step k' with $\text{STATE} = \text{SYNC2}$, does so with $est = d$. Notice that every process which decides at line 21 in step $k' + 1$ (that includes p_w), has only messages with $\text{STATE} = \text{SYNC2}$ in the $msgSet[k' + 1]$, and hence, all these messages have $est = d$. Thus, any process which decides in line 21 of step $k' + 1$, sets its est to d in line 19, and then decides on its $est = d$ in line 21. It follows that every process which decides in step $k' + 1$ decides d. (Recall that no process can decide in line 15 of step $k' + 1$.) It remains to be shown that no process decides a different value in a higher step of sn or in a higher session.

From line 20 we have $|Halt_w[k'+1]| \leq t$, and hence, Lemma 17 implies that $msgSet_w[k'+1]$ contains at least $n-t$ messages, i.e., messages from a majority of processes. Furthermore, the last condition in line 20 requires that every message in $msgSet_w[k' + 1]$ has $\text{STATE} = \text{SYNC2}$. Thus, a majority of the processes send a message with $\text{STATE} = \text{SYNC2}$ in step $k'+1$.

As the round $k'+1$ message from process p_v has $\text{STATE} = \text{SYNC2}$ and $est = d$, by applying Claim 19, it follows that for messages in round $k'+1$, (1) every message with $\text{STATE} = \text{SYNC2}$ in round $k'+1$ has $est = d$, and (2) every message with $\text{STATE} = \text{SYNC1}$ has $est \geq d$.

Now consider the est value of any process p_i at the end of step $k'+1$. If $\text{STATE}_i[k'+1] = \text{NSYNC}$, then p_i has received at least one message with $\text{STATE} = \text{SYNC2}$ and $est = d$ (because a majority of processes have sent such messages and, in every step, p_i receives messages from at least $n-t$ processes, a majority), and therefore, updates its est to d (line 28). If $\text{STATE}_i[k'+1] \in \{\text{SYNC1}, \text{SYNC2}\}$ then $Halt_i[k'+1] \leq t$ (line 22 and line 24). Therefore, $msgSet_i[k'+1]$ contains at least $n-t$ messages (Lemma 17). Furthermore, $msgSet_i[k'+1]$ contains no message with $\text{STATE}_i[k'+1] = \text{NSYNC}$ (line 17 and line 18). Therefore, from Claim 19, every message in $msgSet_i[k'+1]$ has $est \geq d$ and at least one message with $\text{STATE} = \text{SYNC2}$ and $est = d$ (because, in step $k'+1$, a majority of processes sent messages with $\text{STATE} = \text{SYNC2}$ and $est = d$). Therefore, in line 19, p_i updates est to d.

Thus every process which completes step $k'+1$ updates its est to d, and every process which decides in step $k'+1$, decides d. Suppose by contradiction that some process decides a value different from d in a higher step of sn or in a higher session. Consider the lowest session sn'' and the lowest step k'' in sn'', in which some process p_j decides a value different from d, say d''. Observe that if p_j decides in line 15 of step k'', then from line 14 it follows that some process has decided d'' in a lower step of sn'' or in a lower session. Thus p_j has decided on its est in line 21. Observe that, given a session sn', the est value of a process at the end of some step $k \geq 2$ is the est value of some process at the end of step $k-1$, and the est value of a process at the end of the step $k=1$ is the est value of some process at the end of step $t+2$ of the previous session $sn'-1$. Therefore, the est value of any process in a step higher than $k'+1$ in session sn, or in a higher session, cannot be different from d. Thus p_j cannot decide d'' in step k'' of sn''; a contradiction. □

Claim 19 *Consider the lowest session sn in which some process decides. If $k'+1 \geq 2$ is the lowest step in sn in which some process decides then: if there are two processes p_x and p_y such that $\text{STATE}_x[k'] \in \{\text{SYNC1}, \text{SYNC2}\}$ and $\text{STATE}_y[k'] = \text{SYNC2}$ then $est_x[k'] \geq est_y[k']$.*

Proof: Suppose by contradiction that there are two processes p_x and p_y such that:

Assumption A1: $\text{STATE}_x[k'] \in \{\text{SYNC1}, \text{SYNC2}\}$, $\text{STATE}_y[k'] = \text{SYNC2}$, $est_x[k'] = c$, $est_y[k'] = d$, and $c < d$.
In the context of session sn, we show Claims 19.1 to 19.7 based on the definition of k', and the Assumption A1. Claim 19.4 contradicts Claim 19.7, which completes the proof of Claim 19 by contradiction. □

Let us define the following sets for $k \in [1, k'+1]$:

- $C[k] = \{p_i | est_i[k] \leq c\}$ (The set of processes that complete step k with $est \leq c$.)
- $crashed[k]$ = the set of processes that crashed before completing step k.
- $NSYN[k] = \{p_i | \text{STATE}_i[k] = \text{NSYNC}\}$.

- $Z[k] = C[k] \cup crashed[k] \cup NSYN[k]$.

Additionally, let us define, $C[0]$ to be the set of processes that start step 1 with est less than or equal to c, $crashed[0]$ to be the set of processes which crash before sending any message in step 1, $NSYN[0] = \emptyset$, and $Z[0] = C[0] \cup crashed[0] \cup NSYN[k]$. We first make the following observation:

Observation A2: $|C[0]| \geq 1$, and hence, $|Z[0]| \geq 1$. Otherwise, if every process starts step 1 with a value greater than c, then $est_x[k'] > c$ (contradicts A1).

Claim 19.1: (1) $\forall k \in [0, k'-1], (crashed[k] \cup NSYN[k]) \subseteq (crashed[k+1] \cup NSYN[k+1])$.
(2) $\forall k \in [1, k']$, if $p_i \notin (NSYN[k] \cup crashed[k])$ then p_i sends messages with STATE $\in$ {SYNC1, SYNC2} in step k, and in all steps lower than k, of this session.
Proof: (1) Suppose by contradiction that there is a process p_i such that $p_i \in crashed[k] \cup NSYN[k]$ and $p_i \notin crashed[k+1] \cup NSYN[k+1]$. Since a crashed process does not recover, $crashed[k] \subseteq crashed[k+1]$, and hence, $p_i \notin crashed[k+1] \cup NSYN[k+1]$ implies that $p_i \notin crashed[k]$. Thus, $p_i \in crashed[k] \cup NSYN[k]$ implies that $p_i \in NSYN[k]$, i.e., p_i completes step k with STATE $=$ NSYNC. Notice that by the definition of k' (i.e., $k'+1$ is the lowest step in which some process decides), the conditions of line 12 and line 14 cannot be true in step $k+1 < k'+1$, for any process. Thus the STATE of p_i remains NSYNC at the end of step $k+1$, i.e., $p_i \in NSYN[k+1]$; a contradiction.

(2) If $p_i \notin (NSYN[k] \cup crashed[k])$ then, from Claim 19.1.1, it follows that $p_i \notin (NSYN[k_1] \cup crashed[k_1])$ for all $k_1 \in [0, k]$; i.e., p_i completes every step lower than or equal to k with STATE $\neq$ NSYNC. Thus p_i has not send any message with STATE $=$ NSYNC in step k or in a lower step. □

Claim 19.2: $\forall k \in [0, k'-1], Z[k] \subseteq Z[k+1]$.
Proof: Suppose by contradiction that there is a process p_i and some $k \in [0, k'-1]$ such that $p_i \in Z[k]$ and $p_i \notin Z[k+1]$. Since $p_i \notin Z[k+1]$, then $p_i \notin crashed[k+1] \cup NSYN[k+1]$. Applying Claim 19.1.1, we get $p_i \notin crashed[k] \cup NSYN[k]$. However, $p_i \in Z[k] = C[k] \cup crashed[k] \cup NSYN[k]$, and hence, $p_i \in C[k]$.

We first observe that p_i sends messages with STATE $\neq$ NSYNC in the first $k+1$ steps: this follows from $p_i \notin crashed[k+1] \cup NSYN[k+1]$ and Claim 19.1.2. As p_i always receives message from itself, and does not send any message with STATE $\neq$ NSYNC in the first $k+1$ steps, so $p_i \notin Halt_i[k+1]$ (line 17). Furthermore, $p_i \notin crashed[k+1] \cup NSYN[k+1]$ implies that p_i completes step $k+1$ with STATE $=$ SYNC1 or STATE $=$ SYNC2. Applying Lemma 17 we have, $p_i \in senderMS_i[k+1]$. Thus, step $k+1$ message from p_i is in $msgSet_i[k+1]$. However, as $p_i \in C[k]$, the step $k+1$ message from p_i contains $est \leq c$. Thus, when p_i evaluates est in line 19 of step $k+1$, p_i considers its own message with $est \leq c$, and hence, adopts a value less than or equal to c as the new est. Thus $p_i \in C[k+1] \subseteq Z[k+1]$; a contradiction. □

Claim 19.3: $\forall k \in [0, k'-1], \forall p_i \notin Z[k+1], Z[k] \subseteq Halt_i[k+1]$.
Proof: Consider a process $p_j \in Z[k]$ and a process $p_i \notin Z[k+1]$. In step $k+1$, $msgSet_i[k+1]$ either contains a message from p_j or does not contain any message from p_j. In the second case,

Lemma 17 implies that $p_j \in Halt_i[k+1]$. Consider the case where $msgSet_i[k+1]$ contains a message m from p_j. From line 17 and line 18, it follows that m contains STATE $\neq$ NSYNC, and hence, $p_j \notin NSYN[k]$. Furthermore, p_j sent a message in step $k+1$, and so, $p_j \notin crashed[k]$. Thus $p_j \notin crashed[k] \cup NSYN[k]$, but we have assumed $p_j \in Z[k]$. So, $p_j \in C[k]$, and hence, m contains an est less than or equal to c. Since $m \in msgSet_i[k+1]$, in step $k+1$, p_i evaluates est to a value less than or equal to c. Thus $p_i \in C[k+1] \subseteq Z[k+1]$; a contradiction. Thus $msgSet_i[k+1]$ does not contain any message from p_j. □

Claim 19.4: $|Z[k'-1]| \leq k'-1$.
Proof: Suppose by contradiction that $|Z[k'-1]| > k'-1$. From Assumption A1, it follows that $p_y \notin Z[k']$. Therefore, from Claim 19.3, $Z[k'-1] \subseteq Halt_y[k']$. Hence, $|Halt_y[k']| > k'-1$. However, $\text{STATE}_y[k'] = \text{SYNC}2$ implies that $|Halt_y[k']| \leq k'-1$ (line 22, line 23); a contradiction. □

Claim 19.5: $p_x \in Z[k']$ and $p_x \notin Z[k'-2]$.
Proof: As $est_x[k'] = c$, so $p_x \in C[k'] \subseteq Z[k']$.

For the second part of the claim, suppose by contradiction that $p_x \in Z[k'-2]$. Then, from Claim 19.3, for every process $p_i \notin Z[k'-1]$, $p_x \in Halt_i[k'-1]$. Therefore, in step k', if any process in $\Pi - Z[k'-1]$ sends a message m, then $p_x \in m.Halt$ (where, $m.Halt$ denotes the $Halt$ field of m). If p_x receives m in step k', then it includes the sender of m in $Halt_x$ (because of condition 2 in line 17), and if p_i does not receive m in step k', then p_i includes the sender of m in $Halt_x$ (because of condition 3 in line 17). Thus $\Pi - Z[k'-1] \subseteq Halt_x[k']$. Using, Claim 19.4, $|Halt_x[k']| \geq |\Pi - Z[k'-1]| \geq n-(k'-1)$. Since $k'+1 \leq t+2$ and $t < n/2$, we have $|Halt_x[k']| \geq n-t > t$. However, $|Halt_x[k']| > t$ implies that $\text{STATE}_x[k'] =$ NSYNC (line 26, line 27); a contradiction. □

Claim 19.6: (1) $\forall k \in [0, k'-3], Z[k] \subset Z[k+1]$. ($Z[k]$ is a proper subset of $Z[k+1]$).
(2) $|Z[k'-2]| \geq k'-1$.
Proof: (1) From Claim 19.2, $Z[k] \subseteq Z[k+1]$ ($k \in [0, k'-1]$). Suppose by contradiction that there is some $g \in [0, k'-3]$ such that $Z[g] = Z[g+1]$.

We first show by induction on the step number k that, for all $k \in [g+1, k'-1]$, $C[k] - (NSYN[k] \cup crashed[k]) \supseteq C[k+1] - (NSYN[k+1] \cup crashed[k+1])$.

Base Case ($k = g+1$): $C[g+1] - (NSYN[g+1] \cup crashed[g+1]) \supseteq C[g+2] - (NSYN[g+2] \cup crashed[g+2])$. Suppose by contradiction that there is a process p_i such that $p_i \in C[g+2] - (NSYN[g+2] \cup crashed[g+2])$ (**Assumption A3**), and $p_i \notin C[g+1] - (NSYN[g+1] \cup crashed[g+1])$ (**Assumption A4**).

Assumption A3 implies that $p_i \notin NSYN[g+2] \cup crashed[g+2]$. Applying Claim 19.1.1, we have $p_i \notin NSYN[g+1] \cup crashed[g+1]$, and therefore, from Assumption A4, it follows that $p_i \notin C[g+1]$. Thus p_i completes step $g+1$ with $est > c$ and STATE $\neq$ NSYNC. Furthermore, Assumption A3 implies that p_i completes step $g+2$ with $est \leq c$ and STATE $\neq$ NSYNC. So, $msgSet_i[g+2]$ contains a message with $est \leq c$ from some process p_j, i.e., $p_j \in senderMS_i[g+2]$ (**Observation A5**). As p_j sends a message with $est \leq c$ in step

$g+2$, it follows that $p_j \in C[g+1] \subseteq Z[g+1]$.

As $p_i \notin NSYN[g+1] \cup crashed[g+1]$ and $p_i \notin C[g+1]$, so from the definition of $Z[g+1]$ we have $p_i \notin Z[g+1]$. Claim 19.3 implies that $Z[g] \subseteq Halt_i[g+1]$. Recall that we assumed $Z[g] = Z[g+1]$ and, from line 17, $Halt_i[g+1] \subseteq Halt_i[g+2]$. Therefore, $Z[g+1] \subseteq Halt_i[g+2]$. Thus $p_j \in C[g+1] \subseteq Z[g+1]$ implies that $p_j \in Halt_i[g+2]$. From Observation A5, $p_j \in senderMS_i[g+2] \cap Halt_i[g+2]$.

As $p_i \notin NSYN[g+2] \cup crashed[g+2]$, so p_i completed step $g+2$ with STATE $=$ SYNC1 or STATE $=$ SYNC2. From Lemma 17 it follows that $senderMS_i[g+2] \cap Halt_i[g+2] = \emptyset$. However, $p_j \in senderMS_i[g+2] \cap Halt_i[g+2]$; a contradiction.

Induction Hypothesis ($k \in [g+1, r]$): $C[k] - (NSYN[k] \cup crashed[k]) \supseteq C[k+1] - (NSYN[k+1] \cup crashed[k+1])$, for some $r < k' - 1$.

Induction Step ($k = r+1$): $C[r+1] - (NSYN[r+1] \cup crashed[r+1]) \supseteq C[r+2] - (NSYN[r+2] \cup crashed[r+2])$. Suppose by contradiction that there is a process p_i such that $p_i \in C[r+2] - (NSYN[r+2] \cup crashed[r+2])$ (**Assumption A6**) and $p_i \notin C[r+1] - (NSYN[r+1] \cup crashed[r+1])$ (**Assumption A7**).

Similar to the base case, applying Assumption A6, A7, and Claim 19.1, gives us $p_i \notin NSYN[r+2] \cup crashed[r+2]$, $p_i \notin NSYN[r+1] \cup crashed[r+1]$, and $p_i \notin C[r+1]$. Thus $p_i \notin Z[r+1]$. Since $g+1 < r+1$, from Claim 19.2, we have $Z[g+1] \subseteq Z[r+1]$, and therefore, $p_i \notin Z[g+1]$.

Applying Claim 19.3 on $p_i \notin Z[g+1]$ implies that $Z[g] \subseteq Halt_i[g+1]$. Recall that we assumed $Z[g] = Z[g+1]$, and from line 17 and $g+1 < r+2$, $Halt_i[g+1] \subseteq Halt_i[r+2]$. Therefore, $Z[g+1] \subseteq Halt_i[r+2]$ (**Observation A8**).

From the induction hypothesis, we have $(C[g+1] - (NSYN[g+1] \cup crashed[g+1])) \supseteq (C[r+1] - (NSYN[r+1] \cup crashed[r+1]))$. From the definition of $Z[g+1]$, $C[g+1] - (NSYN[g+1] \cup crashed[g+1]) \subseteq C[g+1] \subseteq Z[g+1]$, and therefore, $C[r+1] - (NSYN[r+1] \cup crashed[r+1]) \subseteq Z[g+1]$. Applying Observation A8, we have $(C[r+1] - (NSYN[r+1] \cup crashed[r+1])) \subseteq Halt_i[r+2]$ (**Observation A9**).

As $p_i \notin Z[r+1]$, p_i completes step $r+1$ with $est > c$ and STATE $\neq$ NSYNC. Furthermore, Assumption A6 implies that p_i completes step $r+2$ with $est \leq c$ and STATE $\neq$ NSYNC. Therefore, $msgSet_i[r+2]$ contains a message with $est \leq c$ from some process p_j, i.e., $p_j \in senderMS_i[r+2]$ (**Observation A10**). As p_j sends a message with $est \leq c$ in step $r+2$, $p_j \in C[r+1] \subseteq Z[r+1]$.

As the step $r+2$ message of p_j is in $msgSet_i[r+2]$, so from line 17 it follows that the message sent by p_j had STATE $\neq$ NSYNC. Therefore, $p_j \notin NSYN[r+1]$, and clearly, $p_j \notin crashed[r+1]$. Therefore, $p_j \in C[r+1] - (NSYN[r+1] \cup crashed[r+1])$. From Observation A9 it follows that $p_j \in Halt_i[r+2]$. From Observation A10, $p_j \in senderMS_i[r+2] \cap Halt_i[r+2]$.

As $p_i \notin NSYN[r+2] \cup crashed[r+2]$ (from Assumption A6), so p_i completed step $r+2$ with STATE $=$ SYNC1 or STATE $=$ SYNC2. Lemma 17 implies that $senderMS_i[r+2] \cap Halt_i[r+2] = \emptyset$. However, $p_j \in senderMS_i[r+2] \cap Halt_i[r+2]$; a contradiction.

From the above result (that we proved by induction), we have $C[k'-2] - (NSYN[k'-2] \cup crashed[k'-2]) \supseteq C[k'] - (NSYN[k'] \cup crashed[k'])$. From Assumption A1, $p_x \in$

$C[k'] - (NSYN[k'] \cup crashed[k'])$). From Claim 19.5, we have $p_x \notin Z[k'-2] \supseteq (C[k'-2] - (NSYN[k'-2] \cup crashed[k'-2])$. In other words, p_x is in $C[k'] - (NSYN[k'] \cup crashed[k'])$ but not in $C[k'-2] - (NSYN[k'-2] \cup crashed[k'-2])$; a contradiction.

(2) Part (1) of this claim implies that for every $k \in [0, k'-3], |Z[k+1]| - |Z[k]| \geq 1$. From Observation A2, $|Z[0]| \geq 1$. Therefore, $|Z[k'-2]| \geq k'-1$. □

Claim 19.7: $|Z[k'-1]| > k'-1$.
Proof: Suppose by contradiction that $|Z[k'-1]| \leq k'-1$. Since $Z[k'-2] \subseteq Z[k'-1]$ (Claim 19.2) and $|Z[k'-2]| \geq k'-1$ (Claim 19.6.2), we have $Z[k'-2] = Z[k'-1]$ and $|Z[k'-2]| = |Z[k'-1]| = k'-1$ (**Assumption A11**).

From Claim 19.5, we know that $p_x \notin Z[k'-2] = Z[k'-1]$. Applying Claim 19.3, we have $Z[k'-2] \subseteq Halt_x[k'-1]$. As $Z[k'-2] = Z[k'-1]$ (from Assumption A11), it follows that $Z[k'-1] \subseteq Halt_x[k'-1]$.

Since $p_x \notin Z[k'-1]$, p_x completes step $k'-1$ with $est > c$ and STATE $\neq$ NSYNC. From Assumption A1, we also know that p_x completes step k' with $est \leq c$ and STATE $\neq$ NSYNC. Therefore, $msgSet_x[k']$ contains a message, say from process p_j, with $est \leq c$, i.e., $p_j \in senderMS_x[k']$. From the definition of $C[k'-1]$, $p_j \in C[k'-1] \subseteq Z[k'-1]$. However, we showed earlier that $Z[k'-1] \subseteq Halt_x[k'-1]$, and from line 17, it follows that $Halt_x[k'-1] \subseteq Halt_x[k']$. Thus $Z[k'-1] \subseteq Halt_x[k']$ and $p_j \in Halt_x[k']$.

From Assumption A1, we know that p_x completed step k' with STATE $=$ SYNC1 or STATE $=$ SYNC2. Therefore, Lemma 17 implies that $senderMS_x[k'] \cap Halt_x[k'] = \emptyset$. However, $p_j \in senderMS_x[k'] \cap Halt_x[k']$; a contradiction. □

9.4 Time-complexity

We now discuss the termination and the time-complexity properties of the algorithm. From the definition of ES_t, for every run R in ES_t, there is an unknown round number GSR from which the system is *synchronous* (property 3 of ES_t). Define *synchronous session* as a session that starts in round GSR or in a higher round. Let sn be the lowest synchronous session, and let f be the number of processes that crash in R.

Lemma 20 *Consider any process p_i that completes step $k \in [1, t+2]$ of session sn. If no correct process has halted before step k, then every process in $Halt_i[k]$ has crashed by step k.*

Proof: Suppose no correct process has halted before step k. For any step $l \in [0, k]$ in sn, let $H[l]$ be the union of all $Halt_j[l]$ such that $Halt_j[l] \neq$ *undefined*. We claim the following which immediately implies the lemma: *Every process in $H[l]$ (for all $l \in [0, k]$) has crashed by step l.*

We prove the claim by induction on step number l. For $l = 0$, the claim is trivially true, because $H[0] = \emptyset$ (base case). Suppose that the claim is true for $l \in [0, l'-1]$ (for some $l'-1 \leq k-1$): every process in $H[l]$ has crashed by step l (induction hypothesis). Consider the set $H[l']$ (induction step). If $H[l'] - H[l'-1] = \emptyset$ then the induction step is trivial.

Suppose by contradiction that there is a process $p_j \in H[l'] - H[l'-1]$ such that p_j has not crashed by step l'. Thus there is a process p_a such that $p_j \notin H_a[l'-1]$ and $p_j \in H_a[l']$.

As p_j has not crashed by step l', no correct process has halted before round k, and sn is a synchronous session, so p_a must have received the step l' message m of p_j. Since, $p_j \in H_a[l']$, m contains either (a) $\text{STATE} = \text{NSYNC}$ or (b) set $Halt_j$ such that $p_a \in Halt_j$. Now, we show both cases to be impossible and thus prove the induction step by contradiction.

From our induction hypothesis, for every step $l'' < l'$, every process in $Halt_j[l'']$ has crashed by step l''. Since no more than t processes can crash in a run, in rounds lower than l', $|Halt_j|$ is never higher than t. Thus p_j can not update its STATE to NSYNC in rounds lower than l' (line 26). Thus the round l' message from p_j does nor contain $\text{STATE} = \text{NSYNC}$.

If the round l' message from p_j contains $Halt_j$ such that $p_a \in Halt_j$ then $p_a \in Halt_j[l'-1] \subseteq H[l'-1]$. However, from our induction hypothesis, every process in $H[l'-1]$ crashes before completing round $l'-1$, which implies that p_a crashes before completing round $l'-1$; a contradiction. □

Lemma 21 *(Time-complexity) In every run of the algorithm in SCS_f, (for any $f \in [0,t]$), every process which decides, decides by round $f+2$.*

Proof: Consider any run R of the algorithm in SCS_f. (Note that, for a run in SCS_f, the first session is synchronous.) If some correct process halted before round $f+2$, then every process receives a DECIDE message (and decides) by round $f+2$. Therefore, suppose by contradiction that no correct process halted before round $f+2$ in R, and some correct process p_i completes round $f+2$ without deciding.

Since at most f processes may crash in R, from Lemma 20, $|Halt|$ at every process is always less than or equal to f. As p_i does not decide in round $f+2$ and $|Halt_i[f+2]| \leq f$, one the following is true: (1) $\text{STATE}_i[f+1] = \text{NSYNC}$, (2) $\text{STATE}_i[f+1] = \text{SYNC1}$, or (3) some process p_j sent a message in round $f+2$ with $\text{STATE} = \text{SYNC1}$. Case 1 requires $|Halt_i| > t$ in round $f+2$ or in a lower round (line 26); a contradiction. Case 2 requires $|Halt_i[f+1]| \geq f+1$, and Case 3 requires $|Halt_j[f+1]| \geq f+1$; a contradiction. □

Lemma 22 *(Termination) Every correct process eventually decides.*

Proof: Suppose by contradiction that some correct process p_i does not decide in a run R. If some correct process halts, then p_i eventually receives its DECIDE message, and decides. Thus, no correct process halts. Consider the lowest synchronous session sn. Since no correct process halts in R, from Lemma 20, $|Halt|$ at every process in session sn is always less than or equal to t (as t is the maximum number of processes that may crash in R).

As p_i does not decide by step $t+2$ of session sn, from line 20, one of the following is true: (1) $\text{STATE}_i[t+1] = \text{NSYNC}$, (2) $\text{STATE}_i[t+1] = \text{SYNC1}$, or (3) some process p_j sent a message in step $t+2$ with $\text{STATE} = \text{SYNC1}$. Case 1 requires $|Halt_i| > t$ in step $t+1$ or in a lower step (line 26); a contradiction. Case 2 requires $|Halt_i[t+1]| \geq t+1$ and Case 3 requires $|Halt_j[t+1]| \geq t+1$; a contradiction. □

9.5 Eventually synchronous results summary

Combining Proposition 16, the global decision lower bounds in [19, 6], and the time-complexity of algorithm A_{es}, we get the following tight bounds in eventually synchronous models:

1. $\forall t \in [1, (n-1)/2]$, $\forall f \in [0, t-3]$, $(ES_t,\ SCS_f,\ \text{UC},\ ld) = f + 2$.
   Local decision bound for uniform consensus.

2. $\forall t \in [1, (n-1)/2]$, $\forall f \in [0, t]$, $(ES_t,\ SCS_f,\ \text{UC},\ gd) = f + 2$.
   Global decision bound for uniform consensus.

10 Concluding Remarks

The time-complexity of local decision is a natural measure in many agreement-based distributed systems. As pointed out in the introduction, in a replication or a transaction system, it may be sufficient for a client to receive the decision value from any process executing the agreement algorithm. Besides, studying local decision metric helps uncover fundamental differences between problems and between models that were not apparent with other metrics. For example, in synchronous models, uniform consensus and non-blocking atomic commit have the same tight bound in terms of global decision, but have different bounds when we consider local decision. Similarly, considering local decision metric allows us to infer that early deciding uniform consensus algorithms are faster in synchronous models than in synchronous runs of eventually synchronous models.

References

[1] Aguilera M. K. and Toueg S., A Simple Bivalency Proof that t-Resilient Consensus Requires $t+1$ Rounds. *Information Processing Letters (IPL)*, 71(3-4):155-158, 1999.

[2] Birman K., Schiper A. and Stephenson P., Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272-314, 1991.

[3] Charron-Bost B. and Le Fessant F., Validity Conditions in Agreement Problems and Time Complexity. *Proc. of 30th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2004)*, Springer Verlag (LNCS 2932), pp. 196-207, 2004.

[4] Charron-Bost B. and Schiper A., Uniform Consensus Harder than Consensus. *Journal of Algorithms*, 51(1):15-37, 2004.

[5] Chandra T. D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM (JACM)*, 43(2):225-267, 1996.

[6] Dutta P. and Guerraoui R., The Inherent Price of Indulgence. *Proc. 21st ACM Symposium on Principles of Distributed Computing, (PODC'02)*, ACM Press, pp. 88-97, Monterey (CA), 2002.

[7] Dutta P., Guerraoui R. and Pochon B., Fast Non-Blocking Atomic Commit: An Inherent Tradeoff. *Information Processing Letters (IPL)*, 91(4):195-200, 2004.

[8] Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 35(2):288-323, 1988.

[9] Dolev D., Dwork C. and Stockmeyer L., On the Minimal Synchronism needed for Distributed Consensus. *Journal of the ACM (JACM)*, 34(1):77-97, 1987.

[10] Dwork C. and Moses Y., Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures. *Information and Computation*, 88(2):156-186, 1990.

[11] Dolev D. and Strong H.R., Authenticated Algorithms for Byzantine Agreement. *SIAM Journal on Computing*, 12(4):656-666, 1983.

[12] Dolev D., Reischuk R and Strong R., Early Stopping in Byzantine Agreement. *Journal of the ACM (JACM)*, 37(4):720-741, 1990.

[13] Fischer M., Lynch N., A Lower Bound for the Time to Assure Interactive Consistency. *Information Processing Letters (IPL)*, 14(4):183-186, 1982.

[14] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2):374-382, 1985.

[15] Guerraoui R., Revisiting the relationship between Non-Blocking Atomic Commitment and Consensus problems. *Proc. 9th International Workshop on Distributed Algorithms, (WDAG'95)*, Springer Verlag (LNCS 972), pp. 87-100, 1995.

[16] Gray J., Notes on database operating systems. *Operating systems; an advanced course*, Springer Verlag (LNCS 60), 1978.

[17] Hadzilacos V., Byzantine agreement under restricted types of failures (not telling the truth is different from telling lies). Tech. Rep. 19-83, Aiken Computation Laboratory, Harvard University, 1983.

[18] Hadzilacos V., On the relationship between the Atomic Commitment and Consensus problems. *Proc. 9th International Workshop on Fault-Tolerant Computing*, Springer Verlag (LNCS 448), pp. 201-208, 1987.

[19] Keidar I. and Rajsbaum S., A Simple Proof of the Uniform Consensus Synchronous Lower Bound. *Information Processing Letters (IPL)*, 85(1):47-52, 2003.

[20] Lynch N., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

[21] Lamport L., The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133-169, 1998.

[22] Lamport L., Lower bounds on consensus. *Unpublished Note*, 2000.

[23] Lamport L. and Fischer M., Byzantine generals and transaction commit protocols. Technical Report 62, SRI International, 1982.

[24] Lamport L., Shostak R. and Pease M., The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382-401, 1982.

[25] Mostefaoui A. and Raynal M., Solving Consensus using Chandra-Toueg's Unreliable Failure Detectors: a general approach. *Proc. 13th International Symposium on Distributed Computing (DISC)*, Bratislava, Slovak Republic, 1999.

[26] Moses Y. and Rajsbaum S., A Layered Analysis of Consensus. *SIAM Journal on Computing*, 31(4):989-1021, 2002.

[27] Moses Y. and Tuttle M.R., Programming Simultaneous Actions using Common Knowledge. *Algorithmica*, 3(1):121-169, 1988.

[28] Pease M., Shostak R. and Lamport L., Reaching Agreement in the Presence of Faults. *Journal of the ACM (JACM)*, 27(2):228-234, 1980.

[29] Skeen D., Nonblocking commit protocols. *ACM SIGMOD International Symposium on Management of Data*, 133-142, 1981.

[30] Skeen D., Crash Recovery in a Distributed Database System. PhD thesis, University of California, Berkeley, May 1982.

[31] Schiper A., Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10(3):149-157, 1997.