THE TIME-COMPLEXITY OF LOCAL DECISION IN DISTRIBUTED AGREEMENT*

PARTHA DUTTA†, RACHID GUERRAOUI‡, AND BASTIAN POCHON§

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 depend 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 is significantly better than previous eventually synchronous consensus algorithms.

Key words. Distributed systems, agreement problems, lower bounds

AMS subject classifications. 68Q25, 68W15

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 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 [20]. 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 [15].

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 [21] (i.e., *global decision*), or even *halt* [6], 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

*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.

†Bell Labs Research India, Lucent Technologies India Ltd., Bangalore-560095, India.

‡Distributed Programming Laboratory, EPFL, CH-1015, Lausanne, Switzerland.

§Distributed Programming Laboratory, EPFL, CH-1015, Lausanne, Switzerland.

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 [27, 22] 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 [23].

In this problem, the processes must decide on a common final value, out of the values they initially proposed, such that all correct processes eventually decide and agree on a common decision. The following algorithm (from [16]) conveys the fact that there can indeed be a difference between local and global decision lower bounds. (Round numbers start from 1.) At the beginning of round 1, process p_1 decides on its proposal value and then sends its decision value to all processes. At the end of every round $i \geq 1$, process p_{i+1} decides on the value contained in the last received message, and if p_{i+1} has not received any message, p_{i+1} decides on its proposal value. Process p_{i+1} then sends its decision value to all processes in round $i+1$. (The correct process with the lowest id, say p_j, succeeds in sending its decision value to all processes in round j. Subsequently, all processes with higher ids decide and propagate the decision value of p_j.) If there are no failures, i.e., $f = 0$, then p_1 decides before sending any message in round 1, and we say that p_1 decides in round 0. In runs of this algorithm with at most $1 \leq f \leq t$ 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 [21] that g_f is exactly $f + 1$. This observation opens several questions.

1. 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.

2. What is the impact of the very nature of the agreement?

(i) 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 $g_f = l_f + 1$?

(ii) 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, 10]. But, do the two problems have the same tight local decision bounds as well?

3. What is the impact of the model? Consider consensus for instance in the eventually synchronous (ES) model [11]. 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.

1. We show in the synchronous model that, except for some specific values of f (which we make 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 [27].) 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, 10], we draw the first line between their time-complexity.

2. 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 equivalent to uniform consensus [16].) 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, 9]. In other words, we show that, for synchronous runs of the 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 [27, 22] and (non-blocking) atomic commit was defined in [15, 29]. The distinction between consensus and uniform consensus, and the relationship with the atomic commit problem were discussed in [18, 16]. Initial lower bound results on the time-complexity of agreement problems were proved in [13], and studied further in [21, 7, 25, 12, 1]. The eventually synchronous model was introduced in [5, 11].

In the synchronous model, one of the initial early halting agreement algorithms was presented in [21]. The early halting lower bound for consensus was proved in [6]. 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 [11]. The equivalence between consensus and uniform consensus in the eventually synchronous model was shown in [16]. Tight bounds for synchronous runs of the eventually synchronous model, in the failure-free case ($f = 0$) was shown in [19, 28, 26], and for the worst-case ($f = t$) was shown in [9]. Techniques that use forward inductions to prove lower bounds on agreement problems were introduced in [24, 1].

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 [24], also used in [19], which we slightly extend to prove local decisions results. 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 create, duplicate or alter messages. However, messages may be lost or reordered. The processes may fail by crashing and do not recover from a crash. The computation proceeds in rounds of message-exchange with round numbers starting from 1 and increasing by 1 in every round. A distributed algorithm A is a collection of deterministic automata, where the automaton for each process executes the following two phases in every round: (a) in the *send phase*, the processes 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. A 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 selected by restricting when processes can crash and specifying which messages are received. A *submodel* of a model M is a model that is a subset of M. We consider the following models.

- For every t such that $0 \leq t \leq n-1$, we define the t-resilient synchronous crash-stop model [23], denoted SCS_t, as follows. In every given run of SCS_t, the following properties hold: (1) if a process starts some round k then it either completes that round or crashes; (2) at most t processes crash; and (3) 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 in that phase, the round k message sent by p_i. (If p_i crashes in the send phase of round k, then there are no delivery guarantees − an arbitrary subset of messages sent by p_i in round k may be lost.)
- For every t such that $0 \leq t \leq n-1$, we define $\mathrm{SCS1}_t$ as the submodel of SCS_t that contains those runs of SCS_t in which at most one process crashes in a round.
- For every 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 every given run of ES_t, the following 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 that completes any round k, receives in that round, the round k messages from at least $n-t$ processes; (3) (*reliable channels*) every message sent by a correct process to a correct process in any round k is received in round k or in a higher round; (4) (*eventual synchrony*) there is

a unknown but finite round number GSR (Global Stabilization Round) such that, in every round $k \geq GSR$, if p_i completes the send phase of the round k, then every process that completes the receive phase of the round, receives in that phase, the round k message sent by p_i. (If p_i crashes in the send phase of round k then, similar to SCS_t, there are no delivery guarantees – an arbitrary subset of messages sent by p_i in round k may be lost.) Also, we say that the run is *synchronous from round GSR*.

Observe that, for every $0 \leq f \leq t \leq n-1$, 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: instead of saying that there is a unique synchronous model, we say that each of the $2n$ models SCS_t and $SCS1_t$ $(0 \leq t \leq n-1)$ 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 [22], denoted NC, the processes start with a proposal value and 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 [15, 29], denoted NBAC, each process casts a vote of whether to abort or commit a transaction, and eventually decides. The termination and the uniform 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 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 consider 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 weak 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 additional 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 additional communication, and hence, our time-complexity lower bounds on UA immediately apply to UC and NBAC problem.

In the synchronous models, 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 [27], denoted IC, each process proposes an initial value and eventually decide on a vector of values. Termination and agreement properties are the same as for uniform consensus. Validity is defined as follows: for every 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 any 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, 6, 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 run 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, for algorithms that solve problem P in model M1, (M1, M2, P, T) is the tight bound for achieving T in submodel M2. The notation captures the common time-complexity tight bounds for agreement problems, where submodel M2 denotes the set of runs (e.g. failure-free runs) for which we want to optimize the algorithms in M1. If we set M2 = M1, the tuple denotes the worst-case bound 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 $a \leq x \leq b$; when $a > b$, $[a, b]$ denotes the emptyset.)

- $\forall t \in [0, n-2]$, (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, 23].
- $\forall t \in [2, n-2], \forall f \in [0, t-1]$, ($SCS_t$, 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 $f+2$ [6].

- $\forall t \in [2, n-1]$, $\forall f \in [0, t-2]$, (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 $t+2$ [9].

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 [24], also used in [19]. We first introduce some definitions and then recall the notion of layering from [24, 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 those in [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 a 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. (Since there are no delayed messages in synchronous models, the channels are empty at the end of every round. Hence, in a synchronous model, we ignore state of channels in configurations at the end of a round.) The state of a process that 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. In this subsection, we consider any given weak binary agreement (WA) algorithm A in model $SCS1_t$. (See Section 2 and Section 3 for a definition of SCS1$_t$ and WA, respectively.)

Extensions in SCS1$_t$. 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 that crashes in round k, and the set of processes that did not receive the round k message from p_i.) In model $SCS1_t$, 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 at least one 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.

Layers. A layer $L(C)$ is the set of configurations defined as $\{C.(i,S) | i \in \Pi, S \subseteq \Pi, (i,S)$ is applicable 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 $k > 0$, $L^k(SC) = L(L^{k-1}(SC))$. (In other words, if SC is a set of round l configurations then $L^k(SC)$ is the set of all round $(l+k)$ configurations that extend any configuration in SC.)

Similar Configurations. Consider a set of round k configurations SC. Two configurations C and D in SC are *similar*, denoted $C \sim D$, if they are identical or they differ at exactly one process. A pair of configurations C and D in SC is *similarly connected* if there are configurations $C = C_0, \ldots, C_m = D$ in SC such that $C_i \sim C_{i+1}$ for every i such that $0 \leq i \leq m-1$. The set SC is *similarly connected* if every pair of configurations in SC is similarly connected. (Our definition of *similarity* does not include the second requirement in the original definition of [19]: there exists a process that is alive in both C and D, and has identical states in C and D. When this property is required in our lower bound proofs, we derive it directly from from our assumption on t and n.)

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.

LEMMA 5.1. *In $SCS1_t$, let $SC = L^0(SC)$ be a similarly connected set of configurations such that in every 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 have crashed in any configuration.*

Proof. 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 that 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 now show that $L^k(SC)$ is similarly connected through the following three claims.

1. *For every configuration $C \in L^{k-1}$(SC), $L(C)$ is similarity connected.* Con-

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

Now we prove our claim. $C.(i,\emptyset) \sim C.(0,\emptyset)$ since the configurations differ only at p_i. If $Q = \emptyset$ then we are done. Hence, let $Q = \{q_1, q_2, \ldots, q_m\}$. For every l in $[1,m]$, let $Q_l = \{q_1, \ldots, q_l\}$, and $Q_0 = \emptyset$. For every l in $[0, m-1]$, $C.(i,Q_l) \sim C.(i,Q_{l+1})$ because the two configurations differ only at q_{l+1}. Thus, $C(i,\emptyset) = C.(i,Q_0)$ and $C1 = C.(i,Q_m)$ are similarly connected.

2. *For every pair of configurations $C, D \in L^{k-1}(\mathrm{SC})$, if $C \sim D$ then $L(C) \cup L(D)$ is similarity connected.* If C and D are identical then the claim immediately follows from claim 1. So consider the case where C and D are distinct. As $C \sim D$, there is a process p_i such that C and D are different only at p_i. Then, configurations $C.(i,\Pi)$ and $D.(i,\Pi)$ are identical because no process receives message from p_i in round k, and p_i has crashed. Hence, $C.(i,\Pi) \sim D.(i,\Pi)$. We know from claim 1 that $L(C)$ and $L(D)$ are each similarity connected. Thus every configuration in $L(C)$ is similarly connected to $C.(i,\Pi)$ and every configuration in $L(D)$ is similarity connected to $D.(i,\Pi)$. As, $C.(i,\Pi) \sim D.(i,\Pi)$, so every configuration in $L(C)$ is similarity connected to every configuration in $L(D)$. Thus, $L(C) \cup L(D)$ is similarly connected.

3. *$L^k(\mathrm{SC})$ is similarity connected.* Consider any pair of configurations $C', D' \in L^k(SC)$. Thus, there are configurations $C, D \in L^{k-1}(SC)$ such that $C' \in L(C)$ and $D' \in L(D)$. As $L^{k-1}(SC)$ is similarity connected, there is a chain of configurations $C = C_0, ..., C_m = D$ such that, for every $l \in [0, m-1]$, $C_l \sim C_{l+1}$. Thus, from claim 2, $L(C_l) \cup L(C_{l+1})$ is similarity connected. A simple induction shows that $L(C_1) \cup ... \cup L(C_m)$ is similarly connected. Thus $C' \in L(C = C_0)$ is similarity connected to $D' \in L(D = C_m)$. As C' and D' are arbitrarily selected from $L^k(SC)$, $L^k(SC)$ is similarity connected.

□

Remarks. The above lemma is a simple generalization of Lemma 2.3 of [19]. 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 $SCS1_t$. Consider any crashed process p_i, and the set of processes J to which messages from p_i were lost in the round in which p_i crashed. Then, in the model of [19], J is only allowed to be a prefix of processes $\{p_1, ..., p_k\}$, whereas in $\mathrm{SCS1}_t$, J is allowed to be any subset of Π.

Informally, the next lemma says that, for any WA algorithm in $SCS1_t$, there are two round f configurations that are almost identical (differ at only one process) but have different decision values in failure-free extensions.

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 5.2. *Consider any WA algorithm A in SCS_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 WA algorithm A in SCS_t. We claim that A solves WA in

$SCS1_t$ as well. A maintains the agreement and termination properties in all runs of $SCS1_t$ because every runs in $SCS1_t$ is also a run in SCS_t. The weak validity property is bit different − it is a condition on the set of failure-free runs. However, observe that the SCS_t and $SCS1_t$ have the same set of failure-free runs. It follows that if A satisfies weak validity property in SCS_t then A also satisfies the property in $SCS1_t$. Thus, A solves WA in $SCS1_t$.

Now consider WA algorithm A in $SCS1_t$. Let C' be any initial configuration of algorithm A and C be the initial 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, such that any two adjacent configurations in the chain are similar. Since C' was arbitrarily selected, the set of initial configurations of A in $SCS1_t$ is similarly connected. From Lemma 5.1 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 z, the round f configuration of $r0$. Similarly, consider any failure-free run $r1$ of A in which correct processes decide 1. We denote by z', the round f configuration of $r1$. Obviously, $val(z) = 0$ and $val(z') = 1$.

As the set of round f configurations of A in $SCS1_t$ is similarly connected, there are some round f configurations of A in $SCS1_t$, $z = y_0, y_1, \ldots, y_m = z'$, 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, $val(z) = val(y_0) = val(y_1) = \ldots = val(y_m) = val(z')$; 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. Consensus. In the following we show a local decision lower bound for weak binary agreement (WA) in synchronous models (SCS_t with $1 \leq t \leq n-1$). 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 5.2 holds when $SCS1_t$ is replaced by 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 at most f crashes) in which every correct process decides in round f or in a higher round.

PROPOSITION 6.1. $\forall t \in [1, n-1]$, $\forall f \in [0, t]$, *(*SCS_t, 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 $f \in [1, t]$. (Also recall that, we define deciding at round 0, as deciding before sending any message in round 1.)

It follows from Lemma 5.2 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 $val(y') = 1$.

As $r(y)$ is a run with at most $f-1$ crashes, it follows from our assumption on A that, in $r(y)$, there is a correct process q_1 that has decided $val(y) = 0$ by round $f-1$. As all correct processes in $r(y)$ are alive in y, it follows that, in y, q_1 is alive and has decided $val(y) = 0$.

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, q_2 has decided $val(y) = 0$ in y. As 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 that 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. □

Incompatibility. It is easy to design a consensus algorithm that matches either the early local decision or the early global decision lower bound. We now show that, maybe surprisingly, no consensus algorithm can match both the early local decision and the early global decision lower bounds, 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 6.2. *$\forall t \in [1, n-2]$, $\forall f \in [0, t-1]$, 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 in SCS_t 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 process decides, and (b) by round $f+1$ of every run with at most $f+1$ failures, some correct process decides.

It follows from Lemma 5.2 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 in round $f+1$ before sending any message, 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.) Also, as $f \leq n-3$, there is a process p_l distinct from p_i and p_j such that, p_l decides 0 and 1 at the end of round $f + 1$ in $r(y_0)$ and $r(y_1)$, respectively.

Now we construct a run r'' by extending configuration y_{1-v}: process p_i crashes in the send phase 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 $r'(y_0)$ because the round f configurations of the two runs differ only at p_i, and p_j does not receive any round $f+1$ message from p_i 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 the agreement property of WA.

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 at most f processes, including p_i, have crashed in y_0, and p_i has not crashed in y_1, it follows that, at most $f - 1$ processes have crashes in y_1. Since $f \leq n - 3$ 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 agreement property of WA. □

6.2. Uniform Consensus. In the following, we show a local decision lower bound for weak binary uniform agreement (UA) in the synchronous models (SCS_t with $1 \leq t \leq n - 1$). Since any uniform consensus (UC) and non-blocking atomic commit (NBAC) algorithm solves UA, the lower bound immediately applies to UC and NBAC. In Section 6.3, we show that the lower bound holds for IC as well.

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

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 5.2 holds when WA and $SCS1_t$ are

replaced by UA and SCS_t, respectively.

PROPOSITION 6.3. $\forall t \in [1, n-1]$, $\forall f \in [0, t-1]$, *(*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.

As every UA algorithm solves WA, it follows from Lemma 5.2 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 that 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 that 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 $r1$ that extends y and in which p_i crashes in round $f+1$ before sending any message. (Recall that $f \leq t-1$.) As p_i has decided 0 in y, it follows from uniform agreement property 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 $r2$ that extends y' and in which p_i crashes in round $f+1$ before sending any message. Notice that no correct process can distinguish $r1$ from $r2$: at the end of round f no alive process that 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 that extends y, $p_i = q_1$ has decided 0 and q_2 has decided 1; a contradiction with uniform agreement.

□

6.3. Non-Blocking Atomic Commit and Interactive Consistency. Recall that the local decision lower bound presented in Section 6.2 holds for UC and NBAC. 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: in Section 7.4 we exhibit a UC algorithm that locally decides in 1 round in failure-free runs.

PROPOSITION 6.4. $\forall t \in [2, n-1]$, *(*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 $C1$ be the initial configuration in which all processes propose 1. Consider the failure-free run $R1$ starting

from $C1$; i.e., $R1 = r(C1)$. 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 cannot decide 0 (and hence, p_i decides 1) in $R1$.

Consider another run $R2$ starting from $C1$, but some process p_j ($\neq p_i$) crashes in round 1 and only p_i receives the round 1 message from p_j. Also, process p_i crashes in round 2, before sending the 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 uniform 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 $t \leq n-1$.

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 the round 1 message from p_j, p_i crashes in round 2 before sending the 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 the 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. But the commit validity property of NBAC requires that no process decides 1 in $R3$ because some process p_j has proposed 0; a contradiction. □

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 easily transformed to a NBAC algorithm (without any additional rounds) as follows. Let $V1$ denote an ordered n-tuple 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 $v \in \{0, 1\}$, then it IC-proposes v. If a process IC-decides $V1$, then is NBAC-decides 1; if the process IC-decides an n-tuple 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 bound in Section 6.2 and Proposition 6.4 applies to IC.

In a related work [10], we show for NBAC algorithms, an incompatibility between globally deciding by round 2 in the failure-free run where all processes propose 1, and globally deciding by round 1 in every run where some process proposes 0. However, that paper does not consider local decisions.

7. A Matching Synchronous Algorithm. In [21], 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. (We do not match the bounds in some boundary cases when f, t, and n are close to each other.) 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 6.2, no single NC algorithm can match both local and global decision lower bounds.)

7.1. IC algorithm overview. Our IC algorithm (Figure 7.1) is inspired by the Byzantine Generals algorithm of [21]. The algorithm runs for at most $t + 1$ rounds.

```
at process p_i:
 1: propose(v_i)
 2: Ordered n-tuples est_i and newest_i: 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}
```

FIG. 7.1. *An early deciding (and halting) interactive consistency algorithm*

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 received 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 estimate (EST) messages containing 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 (DEC) 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 some 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 its current est_i value, provided p_i ensures that all other processes see its current est_i. 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 sent that *est* to all processes in round k; so p_i can decide at the end of that round.

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. (We

omit the subscript of the variable when we make a statement that applies to multiple processes.) For $1 \leq r \leq t+1$, $faulty^r$ denotes the set of processes that have crashed by round r, and $faulty^0$ equals $\emptyset$. For any pair of ordered n-tuples 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) $d \not\preceq d'$ if $d \preceq d'$ is false.

First, we make the following simple observations that we frequently use: (1) (**Observation O1**) for the *est* value at every process and every $j \in [1,n]$, $est[j]$ is either the proposal value of p_j or $\perp$, (2) (**Observation 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 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). Thus we only detail the proof of uniform agreement. We start with some general lemmas about the algorithm.

LEMMA 7.1. *If for some $r \in [1,t]$ no process decides by round r, then the following holds for every process p_i that completes round r. If $lastRound_i^r = true$, then every 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 $r \in [1,t]$.

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, every process p_j that 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 every process p_i that completes round k. If $lastRound_i^k = true$, then every 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 that complete round k, we have $newest^k \preceq d'$. 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 7.2. *If a process p_i does not halt or crash by round $r \in [0,t]$, then p_i has $halt_i^k \neq newhalt_i^k$ for all $k \in [1, r-1]$.*

Proof. Obvious from the algorithm. □

LEMMA 7.3. *If no correct process halts by some round $r-1 \in [0, t-1]$, 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. (For uniformity of presentation, we slightly abuse the terminology and say that *for all runs, no process halts or crashes by round 0.*) Suppose there is a round r such that no correct 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, $halt_i^{r'} = newhalt_i^{r'-1} \subseteq newhalt_i^{r'}$. Thus $|newhalt_i^r| \geq r$. Every process in $newhalt_i^r$ has either halted by round $r-1$ or crashed by round r. Since no correct process halts by round $r-1$, $newhalt_i^r \subseteq faulty^r$, and hence, $|faulty^r| \geq r$. □

LEMMA 7.4. *If no correct 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 $r+1 \in [2,t]$. Suppose that no correct process halts by round $r+1$. Consider any correct process p_i. Since p_i does not halt by round $r+1 \leq t$, it follows from Lemma 7.2 that for $r' \in [1,r]$, $halt_i^{r'} \neq newhalt_i^{r'}$. Since no correct process halts by round $r-1 \leq t-1$, applying Lemma 7.3, we have $|faulty^r| \geq r$. □

LEMMA 7.5. *If every process that decides, decides in line 29 of round $t+1$ or line 33 of round $t+1$, then $|faulty^t| = t$.*

Proof. The proof is trivial when $t = 0$. Thus we consider the case $t \geq 1$. Suppose that every process that decides, decides in line 29 of round $t+1$ or line 33 of round $t+1$. 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, from Lemma 7.2 it follows that for every $g \in [1, t-1]$, $newhalt_i^g \neq halt_i^g$. Thus for every $g \in [1,t]$, $newhalt_i^g \neq halt_i^g$. Since no process decides (and hence, halts) by round t, by applying Lemma 7.3 (with $r-1 = t-1$), we have $|faulty^t| \geq t$. As at most t processes can crash in a run, $|faulty^t| = t$. □

LEMMA 7.6. *(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. Let p_i be a process 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 of round $t+1$:* Thus, process p_i decides either in (1) line 18 or (2) line 29 of round $r \leq t+1$. In both cases, we show the following: no process can decide an n-tuple different from d in round r, and any process that completes round r without deciding in line 18 and line 29, does so with $newest^r = d$. This implies uniform agreement because every process that decides in round r has decision value same as its $newest^r$, and in subsequent rounds, d is the only surviving

newest and *est* value. (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 7.1 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 the send phase of 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: Thus $est_i = newest_i$ is d in line 28 of round r, and p_i sent (r, EST, d) in 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 would not have executed 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 (as p_j completes the send phase of round r, none of its messages are lost); a contradiction. Suppose that p_j decides in line 29. Thus $est_j = newest_j$ is $d1$ in line 28 of round r, and p_i 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., $d = d1$. If p_j decides in line 33, then it decides on the *newest* value adopted in round $t + 1$. We show below that every process that updates its *newest* in round k, updates it to d.

We now show that any process that completes round r without deciding in 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 and line 29. Process p_j updates its variable *newest* in line 21 or line 26. Suppose p_j updates 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 in 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 $d2 \not\preceq d$. Consequently, there is a process p_m such that p_j receives $d3 \not\preceq d$ 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 in 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 that 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 that decides in round $t + 1$, decides the same value. Therefore, we need to only consider the case where every process that decides, decides at line 33 of round $t + 1$. From Lemma 7.5, we have $|faulty^t| = t$. Hence, every process that enters round $t + 1$, is a correct process. Consequently, every process that 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 that 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 of our IC algorithm. We show through the following lemma that, in runs with at most $f \geq 1$ 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 6.4 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 round 1 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. Also, 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 $est[l] \neq \perp$. On the other hand, if p_j sends an *est* such that $est[l] \neq \perp$ in round k then we 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 in a round.) Clearly, if p_i propagates l in round k, then p_i learns l in a lower round.

LEMMA 7.7. *In every 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 halts by round $f+1 \leq t$. It follows from Lemma 7.4 that $|faulty^f| \geq f$. Since at most f processes crash in the run, $|faulty^f| = f$ and every process that enters round $f+1$ is correct. Furthermore, since no correct process halts by round f, Lemma 7.4 implies that $|faulty^{f-1}| \geq f-1$. Since $|faulty^f| = f$, *at most* one process crashes in round f.

Let S be the set of processes that 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 decides in 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. Suppose by contradiction that, at some process in S, $lastRound = true$ in line 16 of round $f+1$. Then that process halts in round $f+1$. This leads to a contradiction because we know that every process in S is correct, and (from our initial assumption) correct processes do not decide (and hence, do not halt) by round $f+1$.

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 that 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 $f+1 \leq t$, from Lemma 7.2 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 by round $f+1$, from Lemma 7.3 (with $r-1 = f \leq t-1$) 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 (in the first paragraph of this proof) that *at most* one process crashes in round f; a contradiction.

(b) Consider a run in which 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 $f+1$. As p_i does not halt by round $f+2 \leq t$, from Lemma 7.2, for every $g \in [1, f+1]$, we have $newhalt_i^g \neq halt_i^g$. Applying Lemma 7.3 (with $r-1 = f \leq t-1$) 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 7.1 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 that completes round 1 has $newest[1] = v_1$ at the end of round 1. At the beginning of round 2, processes set est to $newest$. Subsequently, at all processes, $newest[1]$ and $est[1]$ are always v_1. Thus, in our transformation of IC algorithm to UC algorithm, 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. $\forall t \in [1, n-1]$, $\forall f \in [0, t]$, (SCS_t, SCS_f, NC, ld) $= f$. Local decision bound for consensus.

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

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

8. Eventually Synchronous Lower Bound. In this section we investigate lower bounds for UC in eventually synchronous models 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 [16]. 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 that 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, i.e., 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, 28, 26], and $t + 2$ rounds in runs with at most $f = t$ crashes [9].

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

PROPOSITION 8.1. *$\forall t$ s.t. $1 \leq t < n/2$, $\forall f \in [0, t-3]$, (ES_t, SCS_f, UC, ld) $\geq f + 2$.*

Remarks. We exclude the following two cases. (1) $t = 0$: in this case, processes can decide after exchanging proposal values in the very first round in synchronous runs (e.g., decide always on the proposal value of p_1). (2) $t \geq n/2$: in this case, we know that there is no UC algorithm in ES_t.

Proof. Suppose by contradiction that there is a UC 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 $f+1$. Since SCS_t is a submodel of ES_t, A solves UC in SCS_t as well. We also observe that any UC algorithm also solves WA. Thus A solves WA in SCS_t. Thus from Lemma 5.2 we know 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 $val(y') = 1$. (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.)

We note that in y or y', any alive process p_j, that is distinct from p_i, has not yet decided. Otherwise, as y and y' differ only at p_i, process p_j would decide the same value v in y and y', and hence, p_j is a correct process that decides v in both $r(y)$ and $r(y')$; a contradiction.

Let z and z' denote the configurations at the end of round $f+1$ of $r(y)$ and $r(y')$, respectively. Runs $r(y)$ and $r(y')$ are runs of A in SCS_t, and hence, synchronous runs of A in ES_t. As at most f processes crash in each run, $r(y)$ and $r(y')$, 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. (We now consider runs of A in ES_t.)

Case 1. $p_i \notin \{q_1, q_3\}$. 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 subcases to consider.

Case 1a. Process p_i is alive in y and y'. Consider the following two synchronous 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 and q_3 receive the message from p_i, (3) q_1 and q_3 crash in round $f+2$ before sending any message, 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_1 and q_3 receive the message from p_i, (3) q_1 and q_3 crash in round $f+2$ before sending any message, 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 1b. 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 synchronous 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 in round $f+2$ before sending any message, 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_1 and q_3 receive the message from p_i, (3) q_1 and q_3 crash in round $f+2$ before sending any message, 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.

Case 2. $p_i \in \{q_1, q_3\}$ and p_i is alive in both y and y'.

Remark. To see why we cannot reuse the proof of Case 1, observe that, if $p_i = q_1$ then run $R1$ is not a valid run of A in SCS_t: in SCS_t, p_i cannot decide in the receive phase of round $f+1$ while some of its message from that round are lost. Similarly, if $p_i = q_3$ then run $R2$ is not a valid run in SCS_t. Hence, in this case, we construct some runs of A in ES_t that are not in SCS_t (i.e., non-synchronous runs), to derive a contradiction.

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. We would like to point out that, as required by the properties of ES_t, in all non-synchronous runs that we construct, we ensure that in every round, processes received at least $n-t$ messages of the current round, and channels are reliable.):

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 in round $f+2$ before sending any message, 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 in round $f+1$ or in a higher round, and (5) no message is delayed. Since $t < n/2 < n-1$, 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 in round $f+2$ before sending any message, 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[1]), (3) if $q_3 \neq p_i$, every process that completes round $f+1$ receives round $f+1$ message from q_3, (4) no process distinct from p_i and q_3 crashes in round $f+1$ or in a higher round, 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 $p_i = q_1$ decides 0 in z). 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, (b) all round $f+1$ messages sent by p_i to processes distinct from p_i and q_3 are lost, and (c) 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 extends y' in the same way as R4 extends y. Namely, R5 is a run such that (1) the round f configuration is y', (2) p_i and q_3 crash in round $f+2$ before sending any message, 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 that completes round $f+1$ receives round $f+1$ message from q_3, (4) no process distinct from p_i and q_3 crashes in round $f+1$ or in a higher round, 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 round $f+1$ message from p_i), and thus, q_3 decides 1 at the end of round $f+1$ in $R5$. 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, (b) all round $f+1$ messages sent by p_i to processes distinct from p_i and q_3 are lost, and (c) 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, (4) no process distinct from p_i crashes in round $f+1$ or in a higher round, 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 $p_i = q_1$ decides 0 in z). 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 8.1 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 weakened synchronous

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

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

 8:     Send phase
 9:     send(s_i, est_i, STATE_i, Halt_i) to all

10:     Receive phase
11:     wait until received messages in round s_i
12:     if STATE_i = DECIDE then
13:         return
14:     if received any (s_i, est′, DECIDE, ∗) then
15:         est_i ← est′; decide(est_i); STATE_i ← DECIDE; go to the next round      {decision}
16:     if STATE_i ∈ {SYNC1, SYNC2} then
17:         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)}
18:         msgSet_i ← { m | m is a round s_i message received from p_j ∉ Halt_i}
19:         est_i ← Min{est | (∗, est, ∗, ∗) ∈ msgSet_i}
20:         if (STATE_i = SYNC2) and (|Halt_i| ≤ t) and (STATE = SYNC2 for every message in msgSet_i) then
21:             decide(est_i); STATE_i ← DECIDE; go to the next round                {decision}
22:         if |Halt_i| ≤ k_i − 1 then
23:             STATE_i ← SYNC2
24:         if k_i ≤ |Halt_i| ≤ t then
25:             STATE_i ← SYNC1
26:         if |Halt_i| > t then
27:             STATE_i ← NSYNC
28:     if (STATE = NSYNC) and (received any (s_i, est′, SYNC2, ∗)) then
29:         est_i ← est′
```

FIG. 9.1. *A uniform consensus algorithm A_{es} in ES_t*

model where the following holds: even 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 message 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 are also valid in the synchronous send-omission model [17] as well as in an asynchronous round based model enriched with a *Perfect* failure detector [2]. Thus the $f + 2$ local decision lower bound in synchronous runs also extends 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 $t \geq n/2$ [11]. As we pointed out earlier, [19, 28, 26] give a UC algorithm in ES_t that matches the global decision bound for synchronous runs with $f = 0$ crashes, and [9] 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 9.1 presents a uniform consensus algorithm A_{es} in ES_t that globally decides (and hence, locally decides) within $f + 2$ rounds in every synchronous run with at most f crashes, for $0 \leq f \leq t$. 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 [9] modified for early decision. A_{es} assumes the following: (1) the model ES_t with $0 \leq t < n/2$ (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. (A matching algorithm that does not rely on each process receiving at least $n - t$ messages in every round is described in [8].)

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 IC algorithm presented in Section 7: 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 message (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 a synchronous model, $Halt_i$ 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 p_j can 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 values, processes also exchange the $Halt$ sets to detect whether the current session is synchronous. Furthermore, to ensure early decision, p_i maintains and exchanges a variable STATE_i which indicates if p_i considers the current session to be synchronous (SYNC1), or if p_i considers the session to be synchronous with the possibility of a decision in the next round (SYNC2), or whether p_i considers the session to be asynchronous (NSYNC).

9.2. Description. The processes invoke propose($*$) with their respective proposal values as a parameter, and the propose 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* (eventual synchrony property 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 in 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, and (4) p_i updates STATE_i to DECIDE upon decision;

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_i received a messages from p_j with $p_i \in Halt_j$;

$msgSet_i$ 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 s_i starts from 1 and est_i is initialized to the proposal value of p_i. Variables STATE_i and $Halt_i$ are initialized to SYNC1 and $\emptyset$ respectively, and if p_i has not yet decided, are reset to their initial values at the beginning of each session. In each round, processes exchange *est*, STATE, and *Halt* variables, update their own variables depending upon the messages received, and possibly decide. In step 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 also includes the set of processes p_j such that: (a) p_i has received an NSYNC message from p_j in step k, (b) p_i has received a message from p_j with $p_i \in Halt_j$ in step k, or (3) p_i has not received any message from p_j in step k.
   - p_i includes in $msgSet_i$ every message received in step 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 the set $Halt_i$, p_i updates STATE_i as follows: if h is lower than the current step number, then STATE_i is set to SYNC2, else if h is at most t then STATE_i is set to SYNC1, otherwise, STATE_i is set to NSYNC.

3. If STATE = NSYNC and p_i receives a message with STATE = SYNC2, then p_i adopts the estimate contained in that 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 every 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* that is distinct from any possible value of the variables in the algorithm. If p_i crashes before completing step k, then $val_i[k] = \textit{undefined}$; if p_i crashes before sending messages in step 1, then $val_i[0] = \textit{undefined}$. For every process p_l that completes step k with $\text{STATE}_l[k] \in \{\text{SYNC1}, \text{SYNC2}\}$, let $senderMS_l[k]$ denote the set of processes that have sent the messages in $msgSet_l[k]$. We first prove the following lemma.

LEMMA 9.1. *Consider any session and a process p_l that completes step k with* $\text{STATE}_l[k] \in \{\text{SYNC}1, \text{SYNC}2\}$. *Then,* $senderMS_l[k] = \Pi - Halt_l[k]$.

Proof. Process p_l completes step k with STATE = SYNC1 or STATE = 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 cases concerning 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 9.2. *(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 9.3. *(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 9.2 CONTINUED.] For now, we assume the above claim, and prove uniform agreement. We later give a proof of Claim 9.3. Suppose that some process p_w decides d at line 21 of step $k' + 1$. From lines 19 and 20 it follows that there is a message in $msgSet_w[k' + 1]$ that has STATE = SYNC2 and $est = d$, say from process p_v. Consider another process p_u that completes step k' with STATE = SYNC2 and $est = d'$. Applying Claim 9.3 twice with $p_x = p_v$ and $p_y = p_u$, and vice-versa, we get $d' = d$. It follows that, every process that completes step k' with STATE = SYNC2, does so with $est = d$. Notice that every process that decides at line 21 in step $k' + 1$ (that includes p_w), has only messages with STATE = SYNC2 in the $msgSet[k' + 1]$, and hence, all these messages have $est = d$. Consequently, every process that 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. Thus, every process that 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 9.1 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 STATE = SYNC2. Thus, a majority of the processes sent a message with STATE = SYNC2 in step $k' + 1$. As the round $k' + 1$ message from process p_v has STATE = SYNC2 and $est = d$, by applying Claim 9.3, it follows that for messages in round $k' + 1$, (1) every message with STATE = SYNC2 in round $k' + 1$ has $est = d$, and (2) every message with STATE = SYNC1 has $est \geq d$.

Now consider the est value of any process p_i at the end of step $k'+1$. If $\text{STATE}_i[k'+1]$ = NSYNC, then p_i has received at least one message with STATE = 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). On the other hand, 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 9.1). Furthermore, $msgSet_i[k'+1]$ contains no message with $\text{STATE}_i[k'+1] = \text{NSYNC}$ (line 17 and line 18). Therefore, from Claim 9.3, every message in $msgSet_i[k'+1]$ has $est \geq d$ and there is at least one message with STATE = SYNC2 and $est = d$ (because, in step $k'+1$, a majority of processes sent messages with STATE = SYNC2 and $est = d$). Therefore, in line 19, p_i updates *est* to d.

Thus every process that completes step $k'+1$ updates its *est* to d, and every process that 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. Again 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 9.3. *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 $\text{STATE}_x[k'] \in \{\text{SYNC}1,\text{SYNC}2\}$ and $\text{STATE}_y[k'] = \text{SYNC}2$ 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 9.3.1 to 9.3.7 based on the definition of k', and the Assumption A1. Claim 9.3.4 contradicts Claim 9.3.7, which completes the proof of Claim 9.3 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 that crash before sending any message in step 1, $NSYN[0] = \emptyset$, and $Z[0] = C[0] \cup crashed[0] \cup NSYN[0]$. We 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 assumption A1).

PROOF SKETCH. Before presenting Claims 9.3.1 to 9.3.7, we give a rough sketch of the overall proof. Recall that Z is the set of processes that either has crashed, has entered state NSYNC, or has estimate less that or equal to c, at the end of a step. $Z[k]$ denotes the set Z at the end of step k. We derive a contradiction on the size of set Z by showing that (1) for p_y to complete step k' with STATE =SYNC2 and $est = d$, we need $|Z[k'-1]| \leq k'-1$, but (2) for assumption A1 to be satisfied, $|Z|$ should increase in every step, and hence, $|Z[k'-1]| > k'-1$.

We first note that, if a process is in set $Z[k]$ then it remains in that set in all higher steps. To see why, note that once a process crashes or enters state NSYNC, it stays in those states. In addition, if a process has $est \leq c$ then, unless it crashes or enters state NSYNC, the process updates its estimate to the lowest estimate seen in that step, which cannot be more than c.

Now, from Assumption A1, process p_y completes step k' with state SYNC2. From the algorithm, this requires that $Halt_y$ set of p_y at the end of step k' is of size at most $k'-1$. Now consider the message from a process p_j in $Z[k'-1]$ to p_y in step k'. Either p_y does not receive a message from p_j, or receives one with state NSYNC, or with $est \leq c$. In the first two cases, p_y puts p_j in its $Halt_y$ set, and the last case is not possible because it requires p_y to update its est to a value lower than d. Thus the set $Z[k'-1]$ is a subset of $Halt_y$ at the end of step k', and hence, $|Z[k'-1]| \leq k'-1$.

From the definition of Z and Assumption A1, process p_x is in $Z[k']$. We also show that p_x is not in $Z[k'-2]$. To see why, assume otherwise. Then p_x sends step $k'-1$ messages with $est \leq c$, and therefore, processes in $\Pi - Z[k'-1]$ do not receive any message from p_x (otherwise, they would update their estimate to a value at most c, and hence be in set $Z[k'-1]$). Note that the number of processes that are in $\Pi - Z[k'-1]$ is more than t as we have already shown $|Z[k'-1]| \leq k'-1 \leq t < n/2$. Thus, in step k', more than t processes send messages with $p_x \in Halt$. From the algorithm, p_x puts all such processes in its $Halt_x$ set. However, a $Halt_x$ set of size more than t requires p_x to enter state NSYNC, a contradiction.

We next show that, at least one process enters the set Z in every step (till step $k'-2$). For ease of presentation, in this proof sketch, we ignore crashed processes and processes with state NSYNC. Suppose by contradiction, no process enters the set Z in some step g; i.e., $Z[g] = Z[g+1]$. Then, arguing as above, processes in $\Pi - Z[g+1]$ do not receive any message from processes in $Z[g]$ (otherwise, they would update their estimate to a value at most c, and hence be in set $Z[g+1]$). It follows from the algorithm that, in subsequent steps, every process in $\Pi - Z[g+1] = \Pi - Z[g]$ ignores estimate values received from any process in $Z[g]$. Thus no process in $\Pi - Z[g]$ adopts an est less than or equal to c. Thus set Z does not change after round g. This contradicts our earlier observation that p_x is in $Z[k']$ but not in $Z[k'-2]$. (The actual proof of this claim is bit involved because we need to consider crashed processes and processes with state NSYNC.)

As $|Z|$ increases by at least 1 in every step till step $k'-2$, we have, $|Z[k'-2]| \geq k'-1$. Using a slightly different argument we can show that $|Z|$ increase by 1 in step $k'-1$ as well. Thus, $|Z[k'-1] > k'-1$, which contradicts our earlier observation. We now give the detailed proof of the claims.

CLAIM 9.3.1: (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 9.3.1.1, it follows that $p_i \notin (NSYN[k_1] \cup crashed[k_1])$ for all $k_1 \leq k$; 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 9.3.2: $\forall k \in [0, k'-1], Z[k] \subseteq Z[k+1]$.

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 9.3.1.1, 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 9.3.1.2. As p_i always receives message from itself, and does not send any message with STATE $\neq$ NSYNC in the first $k+1$ steps, it follows that $p_i \notin Halt_i[k+1]$ (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 9.1 we have, $p_i \in senderMS_i[k+1]$. Thus, the 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 $est_i[k] \leq c$. Thus, when p_i evaluates est in line 19 of step $k+1$, p_i considers its own message with $est_i[k] \leq c$, and hence, adopts a value less than or equal to c as $est_i[k+1]$. Thus $p_i \in C[k+1] \subseteq Z[k+1]$; a contradiction. □

CLAIM 9.3.3: $\forall k \in [0, k'-1], \forall p_i \notin Z[k+1], Z[k] \subseteq Halt_i[k+1]$.

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 9.1 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 has 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, message m from p_j 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 9.3.4: $|Z[k'-1]| \leq k'-1$.

Proof. From Assumption A1, it follows that $p_y \notin Z[k']$. Therefore, from Claim 9.3.3, $Z[k'-1] \subseteq Halt_y[k']$. On the other hand, $\text{STATE}_y[k'] = \text{SYNC2}$ implies that $|Halt_y[k']| \leq k'-1$ (line 22 and line 23). Thus, $|Z[k'-1]| \leq k'-1$. □

CLAIM 9.3.5: $p_x \in Z[k']$ and $p_x \notin Z[k'-2]$.

Proof. As $est_x[k'] = c$, we have $p_x \in C[k'] \subseteq Z[k']$.

For the second part of the claim, suppose by contradiction that $p_x \in Z[k'-2]$. Then, from Claim 9.3.3, 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 9.3.4, $|Halt_x[k']| \geq |\Pi - Z[k'-1]| \geq n - (k'-1)$. 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 $\text{STATE}_x[k'] = \text{NSYNC}$ (line 26 and line 27); a contradiction. □

CLAIM 9.3.6: (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 9.3.2, $Z[k] \subseteq Z[k+1]$ ($k \in [0, k'-1]$). 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 $k \in [g+1, k'-1]$, $C[k] - (NSYN[k] \cup crashed[k]) \supseteq C[k+1] - (NSYN[k+1] \cup crashed[k+1])$. (This statement corresponds to the brief argument presented in the proof sketch where we showed that if we ignore crashed processes and processes with NSYNC state, then the set Z does not increase after step g.)

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 9.3.1.1, 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 $\text{STATE} \neq \text{NSYNC}$. Furthermore, Assumption A3 implies that p_i completes step $g+2$ with $est \leq c$ and $\text{STATE} \neq \text{NSYNC}$. 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 $p_i \notin C[g+1]$, from the definition of $Z[g+1]$ we have $p_i \notin Z[g+1]$. Claim 9.3.3 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 $p_i \notin NSYN[g+2] \cup crashed[g+2]$, it follows that p_i completed step $g+2$ with $\text{STATE} = \text{SYNC1}$ or $\text{STATE} = \text{SYNC2}$. From Lemma 9.1 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 9.3.1, 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 9.3.2, we have $Z[g+1] \subseteq Z[r+1]$, and therefore, $p_i \notin Z[g+1]$.

Applying Claim 9.3.3 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 $msgSet_i[r+2]$, 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), p_i completed step $r+2$ with STATE $=$ SYNC1 or STATE $=$ SYNC2. Lemma 9.1 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 9.3.5, 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 9.3.7: $|Z[k'-1]| > k'-1$.

Proof. Suppose by contradiction that $|Z[k'-1]| \leq k'-1$. Since $Z[k'-2] \subseteq Z[k'-1]$ (Claim 9.3.2) and $|Z[k'-2]| \geq k'-1$ (Claim 9.3.6.2), we have $Z[k'-2] = Z[k'-1]$ and $|Z[k'-2]| = |Z[k'-1]| = k'-1$ (**Assumption A11**).

From Claim 9.3.5, we know that $p_x \notin Z[k'-2] = Z[k'-1]$. Applying Claim 9.3.3, 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 9.1 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 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* (eventual synchrony property 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 9.4. *Consider any process p_i that completes step $k \in [1, t+2]$ of session sn. If no correct process decides before step k, then every process in $Halt_i[k]$ has crashed by step k.*

Proof. Suppose no correct process decides before step k in session sn. For every 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 $l \in [0, k]$) crashes 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 $H[l]$ crashes 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 $p_j \notin Halt_a[l'-1]$ and $p_j \in Halt_a[l']$.

As p_j has not crashed by step l', no correct process has decided before round k, and sn is a synchronous session, so p_a must have received the step l' message m of p_j. Since, $p_j \in Halt_a[l']$, m contains either (a) STATE = 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 not contain STATE = 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 $H[l'-1]$ has crashed before completing round $l'-1$, which implies that p_a has crashed before completing round $l'-1$; a contradiction. □

LEMMA 9.5. *(Time-complexity) In every run of the algorithm in SCS_f, (for any $f \in [0, t]$), every process that 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 decides by round $f+1$, then every process receives a DECIDE message (and decides) by round $f+2$. Therefore, suppose by contradiction that no correct process decides by round $f+1$ 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 9.4, in every round, $|Halt|$ at every alive process is 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 other process p_j sent a message in round $f+2$ with STATE $=$ SYNC1. Case 1 requires $|Halt_i| > t$ in round $f+1$ or in a lower round (line 26); a contradiction. Case 2 and 3 is not possible because $|Halt[f+1]| \leq f$ at p_i and p_j, and therefore, p_i and p_j sets STATE to SYNC2 in round $f+1$. □

LEMMA 9.6. *(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 decides, then every correct process receives a DECIDE message and decides. Thus, no correct process decides. Consider the lowest synchronous session sn. Since no correct process decides in R, from Lemma 9.4, in every step, $|Halt|$ at every alive process in session sn is 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 other process p_j sent a message in step $t+2$ with STATE $=$ SYNC1. Case 1 requires $|Halt_i| > t$ in step $t+1$ or in a lower step (line 26); a contradiction. Case 2 and 3 is not possible because $|Halt[t+1]| \leq t$ at p_i and p_j, and therefore, p_i and p_j sets STATE to SYNC2 in round $t+1$. □

9.5. Eventually synchronous results summary. Combining Proposition 8.1, the global decision lower bounds in [19, 9], 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 decisions is a natural measure in many agreement-based distributed systems. As pointed out in the introduction, in a replication or a transactional system, it may be sufficient for a client to receive the decision value from any process executing the agreement algorithm. Besides, studying the local decision metric helps uncover fundamental differences between problems and between models that were not apparent with other metrics. For example, in a synchronous model, 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 a local decision metric allows us to infer that early deciding uniform consensus algorithms are faster in a synchronous model than in synchronous runs of an eventually synchronous model.

REFERENCES

[1] MARCOS KAWAZOE AGUILERA AND SAM TOUEG, *A simple bivalency proof that t-resilient consensus requires t+1 rounds*, Inf. Process. Lett., 71 (1999), pp. 155–158.

[2] TUSHAR DEEPAK CHANDRA AND SAM TOUEG, *Unreliable failure detectors for reliable distributed systems*, J. ACM, 43 (1996), pp. 225–267.

[3] BERNADETTE CHARRON-BOST AND FABRICE LE FESSANT, *Validity conditions in agreement problems and time complexity*, in SOFSEM, vol. 2932 of Lecture Notes in Computer Science, Springer, 2004, pp. 196–207.

[4] Bernadette Charron-Bost and André Schiper, *Uniform consensus is harder than consensus*, J. Algorithms, 51 (2004), pp. 15–37.
[5] Danny Dolev, Cynthia Dwork, and Larry J. Stockmeyer, *On the minimal synchronism needed for distributed consensus*, J. ACM, 34 (1987), pp. 77–97.
[6] Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong, *Early stopping in byzantine agreement*, J. ACM, 37 (1990), pp. 720–741.
[7] Danny Dolev and H. Raymond Strong, *Authenticated algorithms for byzantine agreement*, SIAM J. Comput., 12 (1983), pp. 656–666.
[8] Partha Dutta, *Time-Complexity Bounds on Agreement Problems*, PhD thesis, Swiss Federal Institute of Technology, Lausanne (EPFL), 2005. Thesis number 3261.
[9] Partha Dutta and Rachid Guerraoui, *The inherent price of indulgence*, Distributed Computing, 18 (2005), pp. 85–98.
[10] Partha Dutta, Rachid Guerraoui, and Bastian Pochon, *Fast non-blocking atomic commit: an inherent trade-off*, Inf. Process. Lett., 91 (2004), pp. 195–200.
[11] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer, *Consensus in the presence of partial synchrony*, J. ACM, 35 (1988), pp. 288–323.
[12] Cynthia Dwork and Yoram Moses, *Knowledge and common knowledge in a byzantine environment: Crash failures*, Inf. Comput., 88 (1990), pp. 156–186.
[13] Michael J. Fischer and Nancy A. Lynch, *A lower bound for the time to assure interactive consistency*, Inf. Process. Lett., 14 (1982), pp. 183–186.
[14] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson, *Impossibility of distributed consensus with one faulty process*, J. ACM, 32 (1985), pp. 374–382.
[15] Jim Gray, *Notes on data base operating systems*, in Advanced Course: Operating Systems, vol. 60 of Lecture Notes in Computer Science, Springer, 1978, pp. 393–481.
[16] Rachid Guerraoui, *Revistiting the relationship between non-blocking atomic commitment and consensus*, in WDAG, vol. 972 of Lecture Notes in Computer Science, Springer, 1995, pp. 87–100.
[17] Vassos Hadzilacos, *Byzantine agreement under restricted types of failures (not telling the truth is different from telling lies)*, Tech. Report 19-83, Aiken Computation Laboratory, Harvard University, 1983.
[18] ———, *On the relationship between the atomic commitment and consensus problems*, in Fault-Tolerant Distributed Computing, vol. 448 of Lecture Notes in Computer Science, Springer, 1986, pp. 201–208.
[19] Idit Keidar and Sergio Rajsbaum, *A simple proof of the uniform consensus synchronous lower bound*, Inf. Process. Lett., 85 (2003), pp. 47–52.
[20] Leslie Lamport, *The part-time parliament*, ACM Trans. Comput. Syst., 16 (1998), pp. 133–169.
[21] Leslie Lamport and Michael J. Fischer, *Byzantine generals and transaction commit protocols*, Tech. Report 62, SRI International, 1982.
[22] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease, *The byzantine generals problem*, ACM Trans. Program. Lang. Syst., 4 (1982), pp. 382–401.
[23] Nancy A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.
[24] Yoram Moses and Sergio Rajsbaum, *A layered analysis of consensus*, SIAM J. Comput., 31 (2002), pp. 989–1021.
[25] Yoram Moses and Mark R. Tuttle, *Programming simultaneous actions using common knowledge*, Algorithmica, 3 (1988), pp. 121–169.
[26] Achour Mostéfaoui and Michel Raynal, *Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach*, in DISC, vol. 1693 of Lecture Notes in Computer Science, Springer, 1999, pp. 49–63.
[27] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport, *Reaching agreement in the presence of faults*, J. ACM, 27 (1980), pp. 228–234.
[28] André Schiper, *Early consensus in an asynchronous system with a weak failure detector*, Distributed Computing, 10 (1997), pp. 149–157.
[29] Dale Skeen, *Nonblocking commit protocols*, in ACM SIGMOD International Conference on Management of Data, 1981, pp. 133–142.