Distributed Computing manuscript No.
(will be inserted by the editor)

Partha Dutta · Rachid Guerraoui · Idit Keidar

The Overhead of Consensus Failure Recovery

Received: date / Accepted: date

Abstract Many reliable distributed systems are consensus-based and typically operate under two modes: a fast *normal* mode in failure-free synchronous periods, and a slower *recovery* mode following asynchrony and failures. A lot of work has been devoted to optimize the normal mode, but little has focused on optimizing the recovery mode. This paper seeks to understand whether the recovery mode is inherently slower than the normal mode.

In particular, we consider consensus algorithms in the round-based eventually synchronous model of [11], where t out of n processes may fail by crashing, messages may be lost, and the system may be asynchronous for arbitrarily long, but eventually the system becomes synchronous and no new failure occurs (we say that the system becomes stable). For $t \geq n/3$, we prove a lower bound of three rounds for achieving a global decision whenever the system becomes stable, and we contrast this with a bound of two rounds when $t < n/3$. We then give matching algorithms for both $t \geq n/3$ and $t < n/3$.

1 Introduction

1.1 Background and motivation

State machine replication [21, 29] is the most popular technique for achieving software fault-tolerance in distributed systems. With this approach, all replicas perform operations that update the data in the same order, and thus remain mutually consistent. In order to agree upon the order of operations, a *consensus* algorithm [24] is often employed, where an instance of consensus is triggered for each user request or group of user requests [22].

In a consensus algorithm, every process proposes a value, and correct processes are required to eventually decide on one of the proposed values, so that no two correct processes decide differently. It is well-known that consensus is not solvable in an asynchronous system even if only one process can crash [14]. On the other hand, it is often unrealistic to assume a completely synchronous system with known time bounds by which all messages arrive. In practice, one can generally assume that the system may behave asynchronously for an arbitrary period of time, but eventually satisfies some timing guarantees. Such system are called *eventually synchronous* [11]. Partially synchronous models [11, 7] and asynchronous models enriched with failure detectors [4] are frequently used to model eventually synchronous systems.

A run in an eventually synchronous system may begin with an unbounded *unstable* period during which failures may occur, no latency bounds are guaranteed to hold, and the output of failure detectors can be arbitrary. However, every run eventually enters a *stable* period, in which latency bounds or guarantees on failure detector outputs do hold, and during which there are no new failures. Many distributed algorithms and systems optimize for stable periods, running a special (more costly) *recovery mode* algorithm upon recovery from unstable periods, and a *normal mode* algorithm while stability lasts. This is true for replication schemes à la Paxos [22, 30, 25]; transaction-based schemes such as [27, 12]; virtually synchronous group communication systems, where the group membership algorithm is run in recovery mode [3, 6, 1]; and also replication engines based on group communication [18, 15, 2].

In this paper, we focus on the cost of the recovery mode. We consider a round-based eventually synchronous model that is close to the crash-stop basic round model in [11], and we are interested in determining time-complexity bounds for consensus algorithms in this model. Obviously, in unstable periods, we cannot bound the number of rounds needed to achieve a global decision (i.e., rounds needed for all correct processes to decide), as this would contradict the FLP re-

P. Dutta
Bell Labs Research, Bangalore, India

R. Guerraoui
School of Computer and Communication Sciences, EPFL, CH 1015, Switzerland

I. Keidar
Department of Electrical Engineering, The Technion, Haifa, 32000, Israel

sult [14]. [1] We can, however, bound the number of rounds needed to reach global decision in stable periods. Specifically, we consider how quickly a consensus algorithm can decide after an arbitrarily long asynchronous failure-prone period, i.e., the cost of recovery of a consensus algorithm from asynchrony and failures. Note that if a system oscillates between unstable and stable periods, this cost of recovery also indicates how long a system has to remain stable in order to guarantee that a consensus algorithm will be able to decide.

1.2 Results and Contributions

We consider an eventually synchronous model in which processes only fail by crashing, and the model ensures that in every run r, there is an unknown round number $GSR(r) \geq 1$, (Global Stabilization Round of run r) such that only correct processes enter round $GSR(r)$,[2] and from that round onwards, messages sent from correct processes to correct processes are received in the same round in which they are sent. (Any message sent before $GSR(r)$ may be lost.) At most t out of n processes may fail in any run. For example, $GSR(r) = 1$ implies that all faulty processes crash before starting round 1 in run r.

Our first result, presented in Section 4, is a lower bound on recovery mode: we show that if $t \geq n/3$, then every consensus algorithm has a run r that requires at least three rounds for global decision from round $GSR(r)$ (i.e., some process decides at or after round $GSR(r) + 2$), for any value of $GSR(r)$. Given the known tight lower bound of two rounds on global decision in runs that are failure-free and stable from the very beginning [19], (also called *nice* runs), we get that there is an inherent overhead of one round for recovering from failures in systems that can be asynchronous when $t \geq n/3$. Intuitively, recovery in the eventually synchronous model is more costly even after the system becomes stable, since an algorithm cannot know that the system has stabilized, and must account for the possibility that processes from which messages do not arrive are in fact correct. Our lower bound is proven by examining a subset of the runs in which each process receives at least $n - t$ messages in each round. Thus, our lower bound also applies to algorithms that wait for $n - t$ messages in each round before starting a new round. (Note that algorithms that do not overcome message loss may wait for $n - t$ messages in each round, but waiting for more messages may violate liveness, as t processes may crash.)

In Section 5, we give a matching consensus algorithm that globally decides by round $GSR(r) + 2$ in every run r, thus showing that our lower bound is tight. This is significantly faster than any previously suggested algorithm. This algorithm also achieves the two-round lower bound in nice runs.

Interestingly, in Section 6, we show that when $t < n/3$, recovery mode is not more costly than the normal mode: we give a consensus algorithm that tolerates $t < n/3$ crashes and globally decides by round $GSR(r) + 1$ in every run r. This suggests that mechanisms such as leases [25,30] and group membership [6], which often slow down the recovery mode in order to expedite the normal mode, are not needed when less than a third of the processes can crash.

2 Related work

In the eventually synchronous model, any algorithm that solves consensus also solves uniform consensus [17], a variant of consensus in which no two processes (whether correct or faulty) are allowed to decide differently. Therefore, for the rest of this paper, wherever we mention consensus, we implicitly refer to its uniform version.

In the synchronous model, the tight bound on the number of rounds for global decision of a uniform consensus algorithm is $t + 1$ [13,5]. But in the eventually synchronous model, there obviously cannot be any bound on the number of rounds for decision, since the system can be asynchronous arbitrarily long. We can, however, bound the number of rounds needed to reach a global decision in stable periods. In [19], it has been suggested to investigate the bound in nice runs of eventually synchronous systems, i.e., those runs that are failure-free and stable from the very beginning. It was shown that the tight bound in such runs is two rounds.

Next, consider synchronous runs in which all crashes are initial, i.e., any process that crashes, crashes before starting round 1. (In our model, this corresponds to the a run r in which $GSR(r) = 1$.) First of all, let us examine the synchronous model. We observe that a simple adaptation of the synchronous consensus algorithm of [23] gives a synchronous uniform consensus algorithm that globally decides in two rounds in every run where all failures are initial. Our lower bound shows that the same performance cannot be achieved in the eventually synchronous model if $t \geq n/3$: in this case, every algorithm has some run r with $GSR(r) = 1$ in which global decision requires three rounds. Thus, our lower bound highlights an inherent difference in time complexity between uniform consensus algorithms for the synchronous model and ones for the eventually synchronous model.

Finally, let us examine recovery from arbitrary periods of asynchrony and failures. The original DLS [11] consensus algorithm for the eventually synchronous model progresses in phases and uses the rotating coordinator approach. Each phase k consists of four rounds, $4k - 3$ to $4k$, and is coordinated by a predefined process. There are runs r in which DLS globally decides only at round $GSR(r) + 2 + 4(t + 1)$: $GSR(r)$ might occur in the second round of a phase and thus "waste" that phase, and the next t phases may be wasted

[1] The time-complexity metric considered in this paper is the number of rounds required for all correct processes to *decide* (global decision). The number of rounds required for all correct processes to *halt* (global halting [8]) may be different.

[2] Our definition of GSR differs from the definition of the Global Stabilization Time (GST) in [11] in that, in the latter, processes may fail after GST.

if they are coordinated by faulty processes. In general, all rotating coordinator algorithms are prone to recovery times linear in t.

Leader-based algorithms can recover from failures faster than rotating-coordinator ones. Roughly speaking, after $GSR(r)$, the phases with faulty coordinators may be prevented if processes elect a leader to coordinate each phase instead of relying on a predefined coordinator [22,26,9]. However, to the best of our knowledge, no previously suggested algorithm meets our bound of three rounds. For example, in Paxos [22], decision can take up to five rounds after $GSR(r)$. More specifically, after a leader fails, three rounds are needed in order to elect a new leader, and then it takes two additional rounds for the leader to achieve consensus. Intuitively, our algorithm achieves the optimal recovery time by running the normal and recovery modes simultaneously.

The leader-based algorithms in [26,9] require that processes receive at least $n - t$ messages in every round, whereas in our model, any number of messages may be lost before $GSR(r)$[3]. This difference is significant in the presence of asynchrony even if there is no message loss, as it may require processes to wait arbitrarily long (for $n - t$ messages) before moving to the next round. This condition does not allow processes to locally advance rounds based only on their clocks. Thus, even if processes' clocks are perfectly synchronized, during periods of asynchrony, a group of fast processes may advance an unbounded number of rounds without reaching decision, while some correct processes may lag behind. In such cases, once synchrony is re-established and the fast processes begin to execute $GSR(r)$, the processes lagging behind may have to execute an unbounded number of rounds (and send and await an unbounded number of messages) in order to catch up. Thus, these protocols have unbounded recovery times. In contrast, if, as in this paper, arbitrary message loss is allowed, then each process can advance rounds according to its local clock, and once all clocks are synchronized (after $GSR(r)$), all process can execute the same round without delay. Moreover, the algorithm in [9] globally decides by round $GSR(r) + 3$, not $GSR(r) + 2$, and the leader-based algorithm of [26] does not achieve the 2-round failure-free lower bound.

In an earlier paper [10], we have considered a slightly different eventually synchronous model, and studied the complexity of consensus algorithms in synchronous runs with failures. We have shown that in runs that are synchronous from the beginning, $t + 2$ is a tight lower bound on the number of rounds for consensus. However, unlike this paper, [10] did not study algorithm complexity in failure-free stable periods that follow unstable (asynchronous and failure-prone) ones, and did not present a protocol that quickly recovers from asynchrony as we do here. The lower bounds presented herein neither imply nor are implied by those in [10]. Furthermore, the $t + 2$ lower bound of [10] holds for any $t \geq 1$, whereas the lower bounds shown in this paper distinguish the cases $t \geq n/3$ and $t < n/3$.

[3] Note that our lower bound proof covers such algorithms as well, because it is restricted to runs in which each process receives $n - t$ messages.

3 Model and Problem Definition

3.1 The eventually synchronous model

We consider a distributed system consisting of a set of $n \geq 3$ processes, denoted by $\Pi = \{p_1, p_2, \ldots, p_n\}$. Every ordered pair of process communicate by message-passing using a communication channel that does not create, duplicate, or alter messages. A communication channel is a set consisting of messages that have been sent but not yet received. Each channel is associated with a single sender and a single destination.

A distributed algorithm A is a collection of deterministic automata, where A_p is the automaton assigned to process p. Each automaton has an initial state. A computation proceeds in *rounds* of message-exchange. Rounds are identified by round numbers that start from 1. At each process, a round consists of three sub-rounds: *send*, *receive*, and *computation*. A sub-round, in turn, consists of an atomic *step*. A step at p atomically does the following: (1) removes a set of messages M (possibly $\emptyset$) from some channels, (2) applies M and the current state st_p of p to A_p, which outputs a new state st'_p and a set of messages to be sent, and then (3) updates the state of p to be st'_p and puts the output messages in respective channels. In particular, a step of a send sub-round puts n messages in the n channels going out from p_i. In a step of a receive sub-round, a process receives some messages but does not send any message. In steps of a computation sub-round, each process computes the messages for the next round, but does not send or receive messages.

Given an algorithm A, a *run* of A is an infinite sequence of sub-rounds of processes such that (1) initially, all channels are empty and every A_p is in its initial state; (2) for each message set M received in a step at a process p, and for every message $m \in M$, the appropriate channel contains m immediately before that step is taken; (3) all steps involving process p are transitions of the state machine A_p; (4) process p executes a sub-round of a round only after executing all lower rounds; and (5) inside a round, sub-rounds are executed in the following order: send, receive, and compute. (Sub-rounds of different processes, possibly at different rounds, may be interleaved.)

In every run, at most a threshold t of the processes may fail by crashing: if some process p_i does not take the assigned steps in some sub-round of a run r, then we say that p_i is *faulty* in r, and p_i does not take any subsequent steps. A process that does not fail in a run r is *correct* in that run. If p_i takes some steps in round k but does not take any step in round $k + 1$, then we say that p_i *crashes in round* k. If p_i does not take any step in round 1, then we say that p_i crashes in round 0, or *crashes initially*. A process *enters* round k if it takes at least one step in round k, and a process *completes*

at process p_i
1: $k \leftarrow 0$; initialize() {*initialize local variables*}
2: **while** true **do**
3: $k \leftarrow k + 1$
4: **for** j = 1 to n **do:** send round k message to p_j
5: receive messages
6: compute() {*compute round* $k + 1$ *messages*}

Fig. 1 A generic algorithm in the eventually synchronous model.

round k if it takes all assigned steps in round k. Note that according to our terminology, a process p_i may complete round k but still crash in round k if it takes all assigned steps in round k but does not take any steps in round $k+1$; in this case, we say that p_i *crashes at the end of round* k. A round k message of process p_i is a message sent by p_i in round k. We say that a message m is *lost* in run r if m is sent but not received in run r.

The eventually synchronous model ensures that the following properties hold in each run r: (1) *self delivery*: in every round, each non-crashed process receives the message from itself; and (2) *eventual synchrony*: there is an unknown but finite round number $GSR(r)$ such that every process that enters round $GSR(r)$ is a correct process, and in every round $k \geq GSR(r)$, each correct process receives a round k message from every correct process.

Observe that any message sent before $GSR(r)$ may be lost, except by its sender. Also note that $GSR(r) = 1$ does not imply that run r is failure-free: it only implies that every process that crashes in r, crashes initially. A run r is called a *nice run* if no process crashes in r and $GSR(r) = 1$. A generic algorithm (modified from [16]) in the eventually synchronous model is shown in Figure 1. A specific algorithm simply describes the initial state assigned in line 1 and the local computation done in line 6.

3.2 Consensus algorithms

In a consensus algorithm, we assume that every process p is provided with two local variables: a read-only variable $prop_p$ and a write-once variable dec_p. In every run r, $prop_p$ is initialized to some value $v \neq \perp$, (we say that p proposes v in r), and dec_p is initialized to $\perp$. We say that p decides d in r if p writes $d \neq \perp$ to dec_p in some step of r. Every run r of a consensus algorithm satisfies the following three properties: (a) (*validity*) if a process decides v then some process has proposed v, (b) (*(uniform) agreement*) no two processes decide differently[4], and (c) (*termination*) every correct process eventually decides.

Consider any consensus algorithm A in the eventually synchronous model. We say that a process p decides in round k of a run of A if p writes a value to dec_p in a step of round k of that run. We say that a run of A achieves global decision at round k if (1) every process that decides in that run decides at round k or at a lower round; and (2) at least one process decides at round k.

[4] Recall that, from [17], every consensus algorithm in the eventually synchronous model also solves uniform consensus. Since this paper focuses on the eventually synchronous model, we consider the uniform variant of consensus.

4 The Lower Bound

In this section, we give a lower bound on the number of rounds for achieving global decision in the eventually synchronous model. In order to strengthen our lower bound, we consider a subset of the runs of the eventually synchronous model satisfying the following two properties: (1) *communication closed rounds*: every message that is sent in a round, and is not received in the same round, is lost and (2) in every round k, each process that completes round k, receives at least $n-t$ round k messages. (Note that we assume these additional properties only for the sake of broadening the scope of our lower bound. The algorithms we present in the ensuing sections do not rely on these properties.)

In addition, since we are concerned with proving a lower bound, without loss of generality, we assume algorithms to be (1) full-information, i.e., a message includes the entire state of the sender, and the state of a process includes all previous steps of the process, (which in turn includes all received messages), and (2) binary, i.e., the proposal values are restricted to 0 and 1.

Definitions and Notation

Consider a run r of a consensus algorithm A. The *round* k *configuration* of r is an ordered n-tuple where element j contains the state of p_j at the *end* of round k in r. (A round 0 configuration, or initial configuration, specifies only the proposal value of each process.) The state of a process that does not complete round k is a special symbol $\top$. The round k configuration of r is *failure-free* if all processes complete round k in r (or there are no initial failures if $k = 0$).

Given a failure-free round k configuration C (of some run r), we define $r_j(C)$ $(1 \leq j \leq n)$ to be a run such that (1) C is the round k configuration of $r_j(C)$; (2) p_j does not enter round $k+1$ (i.e., p_j crashes at the end of round k); and (3) $GSR(r_j(C))$ is $k+1$. Note that the run $r_j(C)$ is unambiguously defined by these three conditions because, (1) as A is a full-information algorithm, C completely defines the run until round k, and (2) the message exchange pattern is completely defined from round $k+1$ onward. We denote by $r^{ff}(C)$ a run such that (1) C is the round k configuration of $r^{ff}(C)$; (2) no process crashes in $r^{ff}(C)$; and (3) $GSR(r^{ff}(C)) = k+1$.

We denote by $val_j(C)$ the decision value of correct processes in $r_j(C)$. We say that a configuration C is *uniFvalent* (uni-failure-valent) if for every pair of processes i, j, $(1 \leq i, j \leq n)$, $val_i(C) = val_j(C)$. We denote this common value by $val(C)$. A uniFvalent configuration is 1-Fvalent if $val(C) = 1$ and 0-Fvalent otherwise. A configuration that is

not uniFvalent is called *biFvalent*. In other words, in a biFvalent configuration, there are two processes p_i and p_j, such that $val_i(C) \neq val_j(C)$. Note that our notion of biFvalency is more restrictive than the traditional notion of bivalency, since the latter is satisfied whenever *any* two extensions of C lead to different decision values, whereas biFvalency requires that two extensions with a specific structure lead to different decision values.

Lower bound proof

Our first lemma shows that the environment (adversary) can cause every algorithm to remain in a biFValent state for an arbitrary number of rounds. A similar result is proven in [28] (for bivalent configurations); we give the proof here for completeness.

Lemma 1 *Let $3 \leq n$ and $1 \leq t \leq n-1$. Let A be a consensus algorithm in the eventually synchronous model. For every $k \geq 0$, there is a failure-free run r in which each process receives at least $n-t$ messages in each round and r's round k configuration is biFvalent.*

Proof : We prove the lemma by induction on round number k.

Base Case: There is a failure-free biFvalent initial configuration. Suppose by contradiction that all initial configurations are uniFvalent. For $0 \leq j \leq n$, let C_j be a failure-free initial configuration in which all processes p_l, where $1 \leq l \leq j$, propose 1, and the rest of the processes propose 0. From validity, $val(C_0) = 0$ and $val(C_n) = 1$. We claim that, for $1 \leq j \leq n$, $val(C_{j-1}) = val(C_j)$. To see why, notice that C_{j-1} and C_j differ only in the proposal value of p_j, and hence, no process can distinguish $r_j(C_{j-1})$ from $r_j(C_j)$. So $val_j(C_{j-1}) = val_j(C_j)$, and since C_{j-1} and C_j are uniFvalent, $val(C_{j-1}) = val_j(C_{j-1}) = val_j(C_j) = val(C_j)$. It follows that if $val(C_0) = 0$ then $val(C_n) = 0$, a contradiction.

Induction Hypothesis: There is a failure-free run r in which each process receives at least $n-t$ messages in each round $1 \ldots k$, and r's round k configuration, C, is biFvalent.

Induction Step: From the induction hypothesis, there is a failure-free biFvalent round k configuration C. Thus, there are $1 \leq i, j \leq n$, such that $val_i(C) = 0$ and $val_j(C) = 1$. Suppose by contradiction that all failure-free round $k+1$ configurations, that extend C and in which each process receives at least $n-t$ messages in round $k+1$, are uniFvalent. For the rest of the proof, we will construct round $k+1$ in which each process receives at least $n-1 \geq n-t$ messages, and hence will be uniFvalent by this assumption.

Let the round $k+1$ configuration of $r^{ff}(C)$ be x-Fvalent ($x \in \{0,1\}$). We show a contradiction assuming $x = 1$. (The case $x = 0$ is similar — in the argument below, we simply use p_j instead of p_i.)

Denote by C^0 the failure-free round $k+1$ configuration that extends C by one round in which all messages sent by p_i are lost and no other message is lost. Note that every process receives $n-1 \geq n-t$ messages in this round. Consider the runs $r_i(C)$ and $r_i(C^0)$. The round $k+1$ configuration of $r_i(C)$ differs from C^0 only in the state of process p_i. Since p_i crashes at the end of round $k+1$ in $r_i(C^0)$, no correct process can distinguish $r_i(C)$ from $r_i(C^0)$. Thus, $val_i(C^0) = val_i(C) = 0$. C^0 being uniFvalent, $val(C^0) = 0$.

We now consider a series of round $k+1$ configurations, each of which extends C by one round. Configuration C^l ($1 \leq l \leq n$) extends C by one round in which (1) no process crashes, and (2) all messages sent by p_i in round $k+1$ are lost except those sent to $\{p_1, ..., p_l\}$. Consider configurations C^{l-1} and C^l. The two configurations differ only at p_l. Thus no correct process can distinguish run $r_l(C^{l-1})$ from $r_l(C^l)$. Thus $val_l(C^{l-1}) = val_l(C^l)$. C^{l-1} and C^l being uniFvalent, $val(C^{l-1}) = val(C^l)$. A simple induction over l, along with our previous observation that $val(C^0) = 0$, gives us $val(C^n) = 0$. Observe that configuration C^n extends C by one round such that no process crashes and no message is lost in round $k+1$. That is, C^n is the round $k+1$ configuration of $r^{ff}(C)$. A contradiction to our assumption that the round $k+1$ configuration of $r^{ff}(C)$ is 1-Fvalent. □

The next lemma shows a lower bound of two rounds, which applies for most values of t. This lemma can also be shown using a simple modification of the proof of [20]. However, a straightforward modification of the proof of [20] would require $t \geq 2$, whereas our proof holds for $t \geq 1$.

Lemma 2 *Let $3 \leq n$ and $1 \leq t \leq n-2$. For every $G \geq 1$, every consensus algorithm has a run r in which every process receives at least $n-t$ messages in each round, $GSR(r) = G$, and some process decides at round $GSR(r)+1$ or at a higher round.*

Proof : Suppose by contradiction that there exists a consensus algorithm B and some round number G, such that for every run r of B in which $GSR(r) = G$, all correct processes decide by round G.

Consider a failure-free run in which every process receives at least $n-t$ messages in each round, and the run's round $G-1$ configuration, C, is biFvalent. (From Lemma 1, such a run exists.) Thus, there are $1 \leq i, j \leq n$ such that $val_i(C) = 0$ and $val_j(C) = 1$. Observe that from our assumption, by the end of round G, every process distinct from p_i decides 0 in $r_i(C)$, every process distinct from p_j decides 1 in $r_j(C)$, and every process decides by the end of round G in $r^{ff}(C)$. Let $x \in \{0,1\}$ be the decision value of processes in $r^{ff}(C)$. We show a contradiction assuming $x = 1$. (The case $x = 0$ is symmetric.)

Consider run $r^{ff}(C')$, where C' is a failure-free round G configuration that extends C by one round, such that in round G, p_i receives its own message, all other messages sent by p_i are lost, and no other message is lost. Then $GSR(r^{ff}(C')) = G+1$. Let p_c be a process distinct from p_i. At the end of round G, p_i cannot distinguish $r^{ff}(C')$ from

$r^{ff}(C)$, and p_c cannot distinguish $r^{ff}(C')$ from $r_i(C)$. Thus, at the end of round G in $r^{ff}(C')$, p_i decides $x = 1$ and p_c decides 0, violating uniform agreement; a contradiction. □

We next prove our three-round lower bound for the special case that $n = 3$ and $t = 1$.

Lemma 3 *Let $n = 3$ and $t = 1$. For every $G \geq 1$, every consensus algorithm has a run r in which every process receives at least $n - t$ messages in each round, $GSR(r) = G$, and some process decides at round $GSR(r) + 2$ or at a higher round.*

Proof : Suppose by contradiction that there exists a consensus algorithm A and some round number G, such that for every run r of A in which $GSR(r) = G$, all correct processes decide by round $G + 1$.

Consider a failure-free run in which every process receives at least $n - t$ messages in each round, and the run's round $G-1$ configuration, C, is biFvalent. (From Lemma 1, such a run exists.) Thus, there are $1 \leq i, j \leq 3$ such that $val_i(C) = 0$ and $val_j(C) = 1$. For convenience of presentation and without loss of generality, we assume that $i = 1$ and $j = 2$.

We consider four runs that extend C. (In each run, note that each process receives at least $n - t = 2$ messages in every round — including one from itself.) Rounds G and $G+1$ of these runs are depicted in Figure 2. We now describe them in words.

- Run a is $r_1(C)$. Thus $GSR(a) = G$, and from our assumption on A, correct processes decide $val_1(C) = 0$ in round $G + 1$.
- Run b is $r_2(C)$. Thus $GSR(b) = G$, and from our assumption on A, correct processes decide $val_2(C) = 1$ in round $G + 1$.
- Run c is $r_3(C1)$, where the round $G+1$ configuration $C1$ is constructed as follows: In round G, the messages from p_1 to $\{p_2, p_3\}$ are lost (this is depicted by the absence of any message arrow from p_1 to $\{p_2, p_3\}$ in round G in Figure 2(c)), and the message from p_2 to p_1 is lost. In round $G + 1$, the messages from p_1 to p_3, and from p_3 to $\{p_1, p_2\}$ are lost. Process p_3 cannot distinguish the round $G + 1$ configuration of run c (i.e., configuration $C1$) from the round $G + 1$ configuration of run a. To see why, notice that p_3 does not receive any message from p_1 in round G and $G+1$ of both runs. Furthermore, p_2 does not distinguish a from c at the end of rounds $G - 1$ and G, and hence, sends identical messages to p_3 in rounds G and $G+1$ of both runs. Therefore, as in run a, p_3 decides 0 in round $G + 1$ in run c. Due to the uniform agreement property, p_1 and p_2 eventually decide 0 in run c.
- Run d is $r_3(C2)$, where the round $G+1$ configuration $C2$ is constructed as follows: In round G, the message from p_1 to p_2 is lost, and the messages from p_2 to $\{p_1, p_3\}$ are lost. In round $G+1$, the message from p_2 to p_3, and from p_3 to $\{p_1, p_2\}$ are lost. Notice that p_3 cannot distinguish the round $G + 1$ configuration of d (i.e., configuration $C2$) from the round $G + 1$ configuration of run b. Therefore, p_3 decides 1 at the end of round $G+1$ in run d. Due to the uniform agreement property, p_1 and p_2 eventually decide 1 in run d.

Now consider runs c and d. At the end of round G, the two runs differ only at process p_3 (because it receives different sets of messages). Processes p_1 and p_2 receive the same set of messages in round $G + 1$ of runs c and d, and they do not include a message from p_3. Therefore, the states of p_1 and p_2 are the same at the end of round $G + 1$ in both runs. Since process p_3 does not send any message after round $G + 1$ (recall that c is $r_3(C1)$ and d is $r_3(C2)$), p_1 and p_2 can never distinguish run c from run d. Therefore, p_1 (and p_2) must decide the same value in c and d: a contradiction. □

Finally, we construct a proof for the general case by simulating a single process with a group of processes.

Lemma 4 *Let $3 \leq n$ and $1 \leq t \leq n - 2$ and $n/3 \leq t$. For every $G \geq 1$, every consensus algorithm has a run r in which every process receives at least $n-t$ messages in each round, $GSR(r) = G$, and some process decides at round $GSR(r)+2$ or at a higher round.*

Proof : We prove this lemma by simulating three processes over a system where $n \geq 3$ and $t \geq n/3$. Divide the set of processes Π into 3 sets of processes, P_1, P_2, and P_3, each of size less than or equal to $\lceil \frac{n}{3} \rceil$. (This is always possible because $3(\lceil \frac{n}{3} \rceil) \geq n$.) Since $t \geq n/3$ and t is an integer, it follows that $t \geq \lceil \frac{n}{3} \rceil$. Therefore, the sets P_1, P_2, and P_3 are each of size less than or equal to t, and hence, in a given run all the processes in any one of the sets may crash.

We now construct runs corresponding to runs with three processes. The relationship between a run r' constructed in this simulation to the corresponding run r with three processes is as follows: (1) if p_i proposes x (0 or 1) in r, then every process in P_i proposes x in r', (2) if p_i crashes without sending any message in some round k of r, then every process in P_i crashes without sending any message in round k of r', (3) if p_i crashes in some round k of r, then every process in P_i crashes in round k of r', (4) if p_i does not crash in r then no process in P_i crashes in r', and (5) for $1 \leq j \leq 3$, if p_i receives a messages from p_j in some round k of r, then every process in P_i receives a message from every process in P_j in round k of r'. (Note that in particular, if p_i does not crash at round k, then it receives a message from itself, and therefore, at round k of r', each process in P_i receives messages from every process in P_i.)

From Lemma 3, every consensus algorithm has a run r in which every process receives at least $n - t$ messages in each round, $GSR(r) = G$, and some process decides at round $GSR(r)+2$ or at a higher round. We simulate r' from r as explained above. Since in r, in each round, each process loses a message from at most one process, in r', each process receives messages from at least $n - t$ processes. Moreover, $GSR(r') = GSR(r)$. Since processes in P_i decide in r'

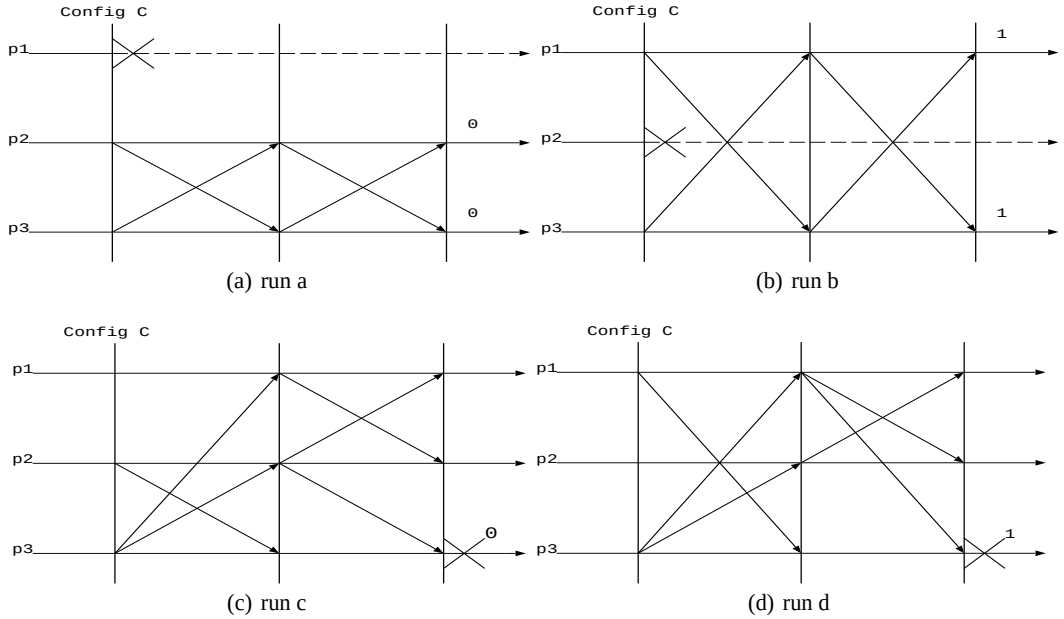

Fig. 2 Rounds G and $G+1$ of the four runs of A.

when p_i decides in r, we get that in r', some process decides at round $GSR(r')+2$ or at a higher round. □

We conclude with the following theorem:

Theorem 1 *Let $3 \leq n$ and $1 \leq t \leq n-2$. For every $G \geq 1$, (a) every consensus algorithm has a run r in which every process receives at least $n-t$ messages in each round, $GSR(r) = G$, and some process decides at round $GSR(r)+1$ or at a higher round.*
(b) if $t \geq n/3$, then every consensus algorithm has a run r in which every process receives at least $n-t$ messages in each round, $GSR(r) = G$, and some process decides at round $GSR(r)+2$ or at a higher round.

Reliable channels

We now consider a stronger eventually synchronous model. We extend the proof to a model where channels are reliable, i.e., all messages from correct processes to correct processes are eventually received. We now argue that Theorem 1 holds with this modification. Our discussion is informal.

If all the runs constructed in the above proofs can be constructed in the modified model then the proofs immediately translate to the modified model. Observe that, the only case when a run in the above proofs cannot be constructed in the modified model is when some message from a correct to a correct process is lost, i.e., the reliable channel property is violated. (Actually, due to the communication closed round restriction assumed in the lower bound proof, any message from a correct process to a correct process, that is not delivered in the same round in which it is sent, will be lost.) We now show how to transform such a run to satisfy the reliable channel property, but without adding any new message. Crucial to our transformation is the property of full-information algorithms that requires any message to contain all lower round messages from its sender to its destination.

Consider any run r in the above proofs in which some message m from a correct process p_i to another correct process p_j is lost (e.g. run c in Lemma 3). Let m be a round k message. Recall that, no message from a correct process to a correct process is lost in round $GSR(r)$ and in higher rounds. Thus, $k < GSR(r)$. Consider the round $GSR(r)$ message m' from p_i to p_j. Message m' contains m because our algorithm is full-information. Thus, on receiving m', process p_j can simulate reception of m in round $GSR(r)$. Similarly, we can simulate the reception of any other lost message from a correct process to a correct process, and thus, satisfy the reliable channel property.

5 A Matching Algorithm for $t < n/2$

We now present a consensus algorithm, UC_1, for the eventually synchronous model with a majority of correct processes, i.e., $t < n/2$. Recall that there is no consensus algorithm in the eventually synchronous model when $t \geq n/2$ [11]. Algorithm UC_1 matches the lower bound of Theorem 1(b) as well as the known lower bound of two rounds in nice runs.

5.1 Algorithm description

Algorithm UC_1 is presented in Figure 3. In every round, each process p_i sends its four primary variables to all processes: (1) the message type $msgType_i$ initialized to

at process p_i

1: $k_i \leftarrow 0$; initialize()
2: **while** true **do**
3: $k_i \leftarrow k_i + 1$
4: **for** j = 1 to n **do:** send round k_i message to p_j
5: receive messages
6: compute()

7: **procedure initialize()**
8: $est_i \leftarrow prop_{p_i}$ {*read the proposal value*}
9: $ld_i \leftarrow p_n$; $ts_i \leftarrow 0$; $msgType_i \leftarrow$ PREPARE; $nextLD_i \leftarrow p_n$; $maxTS_i \leftarrow 0$
10: round 1 message $\leftarrow (1, msgType_i, est_i, ts_i, ld_i)$

11: **procedure compute()**
12: **if** $dec_{p_i} = \perp$ **then**
13: $nextLD_i \leftarrow p_j$ where $j = \textbf{Max}\{w|$ received a round k_i message from $p_w\}$
14: $maxTS_i \leftarrow \textbf{Max}\{ts|$ received a message $(k_i, *, *, ts, *)\}$
15: **if** received $(k_i$, DECIDE, $est', ts', *)$ **then**
16: $est_i \leftarrow est'$; $ts_i \leftarrow ts'$; $dec_{p_i} \leftarrow est_i$; $msgType_i \leftarrow$ DECIDE {*decision*}
17: **else if** received $(k_i$, COMMIT, $*, *, *)$ from a majority of processes (including p_i) and ld_i **then**
18: $dec_{p_i} \leftarrow est_i$; $msgType_i \leftarrow$ DECIDE {*decision*}
19: **else if** (received$(k_i, *, *, *, ld_i)$ from a majority of processes) {***COMMIT-1***}
and (received $(k_i, *, *, maxTS_i, ld_i)$ from ld_i) {***COMMIT-2***}
and $(ld_i = nextLD_i)$ **then** {***COMMIT-3***}
20: $msgType_i \leftarrow$ COMMIT; $est_i \leftarrow est$ received from ld_i; $ts_i \leftarrow k_i$
21: **else**
22: $est_i \leftarrow$ any est s.t. received $(k_i, *, est, maxTS_i, *)$; $ts_i \leftarrow maxTS_i$; $msgType_i \leftarrow$ PREPARE
23: $ld_i \leftarrow nextLD_i$
24: round $k_i + 1$ message $\leftarrow (k_i + 1, msgType_i, est_i, ts_i, ld_i)$

Fig. 3 Algorithm UC_1.

PREPARE, (2) an estimate est_i of the decision value, initialized to the proposal value (read from $prop_{p_i}$), (3) the timestamp ts_i of the estimate value, initialized to 0, and (4) the leader ld_i of the current round, initialized to p_n. In the computation sub-round, processes update their primary variables depending on the messages received in that round, and possibly decide.

We now briefly explain the purpose of these variable at process p_i. Roughly speaking, the message type indicates the level of progress a process has made towards reaching a decision. In the computation sub-round of round k, if p_i sees a possibility of decision in the next round, then it sends a round $k+1$ message with type COMMIT. We then say that p_i *commits in round* k. Once the process decides, it sends messages with type DECIDE in all subsequent rounds. Otherwise, the message type is PREPARE.

In the computation sub-round of a round k in which p_i does not decide and has not yet decided, p_i adopts one of the estimate values received in that round. Process p_i also adopts the timestamp received along with the estimate, unless p_i commits in round k, in which case p_i updates its timestamp to k. Thus, the timestamp associated with an estimate value x simply indicates a round number in which some processes has committed while adopting estimate x.

The *leader of* p_i *at round* $k \geq 2$ is simply the process p_j with the highest id from which p_i received a round $k-1$ message. Process p_n is the leader at all processes in round 1. Note that different processes may have different leaders in the same round.

We now describe the computation sub-round in more detail. Once a process p_i decides, it sends a DECIDE message with the decision value in every round. Otherwise, in round k, p_i updates its primary variables as follows. From the set of messages received, p_i first computes its leader for the next round ($nextLD_i$) and the highest timestamp received ($maxTS_i$). Then it executes the following four conditional statements. (A statement is executed only if the conditions in all the previous statements are false.)

- If p_i receives a DECIDE message then it decides on the received estimate (by writing that estimate in dec_{p_i}).
- If p_i receives COMMIT messages from a majority of processes, including itself and its current leader, then p_i decides on its own estimate.
- Let ld_i be the leader of p_i at round k. Consider the following three conditions on the messages received by p_i: *commit-1*: received messages from a majority of processes that say that ld_i is their leader at round k; *commit-2*: received a message from ld_i that has the highest timestamp ($maxTS_i$) and has ld_i as the leader; and *commit-3*: $ld_i = nextLD_i$. If all three conditions are satisfied, then p_i sets its message type (for the round $k+1$ message) to COMMIT, adopts the estimate received from ld_i, say x, and sets its timestamp to the current round number k. We say that p_i *commits in round* k *with estimate* x.
- Otherwise, p_i adopts the estimate and the timestamp of the message with the highest timestamp $maxTS_i$, and sets its message type to PREPARE.

Finally p_i updates its ld_i to $nextLD_i$ and composes the message for the next round.

5.2 Correctness of the consensus algorithm UC_1

Lemma 5 *Until a process decides, its timestamp is non-decreasing.*

Proof : If a process p_i does not decide in round k, then it can change its timestamp by adopting either k, or the maximum timestamp ($maxTS$) received in messages of round k, as its new timestamp. Since, p_i receives its own message in every round, $maxTS$ is never lower than its current timestamp. Also, a simple induction shows that the timestamp of a process is always less than or equal to its round number. Thus when a process updates its timestamp, the new timestamp value is greater than or equal to the old value. □

Lemma 6 *For every round k, no two processes commit with different estimates in round k, and no two processes commit with different $newLD$s in round k.*

Proof : Consider two processes p_i and p_j that commit in round k with estimates est_i and est_j, and $newLD$ values $newld_i$ and $newld_j$, respectively. Also, in round k, let ld'_i be the leader of p_i and ld'_j be the leader of and p_j. Thus, from commit-1, each of them has received in round k a majority of messages that contain ld'_i and ld'_j as leaders, respectively. As two majorities intersect, $ld'_i = ld'_j$. Furthermore, from commit-3, $newld_i = ld'_i$ and $newld_j = ld'_j$. So, $newld_i = ld'_i = ld'_j = newld_j$.

From the algorithm, p_i commits with the estimate sent by ld'_i, and p_j commits with the estimate sent by ld'_j. As $ld'_i = ld'_j$, p_i and p_j commit with same estimate. □

Lemma 7 *For every round k, all round k messages with $msgType =$ COMMIT have identical estimate values and identical ld values.*

Proof : Immediate from Lemma 6. □

Lemma 8 *If some process sends a message with timestamp $ts > 0$ and estimate x then some process commits in round ts with estimate x.*

Proof : If a process p_i sends a message with timestamp ts then p_i sets its timestamp to ts in some round. Consider the lowest round k in which some process p_j sets its timestamp to ts. From the definition of k, p_j cannot receive ts from another process in round k. Thus p_j commits with timestamp ts in round k, and from the algorithm, $k = ts$.

Also, from the algorithm, if a process adopts a timestamp from a message, it also adopts the associated estimate. Therefore, by induction on rounds of the run, we can show that each estimate is associated with a ts equal to a round number in which it was committed. □

Lemma 9 (Uniform Agreement) *No two processes decide differently.*

Proof : If no process ever decides then the lemma trivially holds. Suppose some process decides. Let k be the lowest round in which some process p_i decides. Process p_i can decide either (1) by receiving a DECIDE message, or (2) by receiving a majority of COMMIT messages, including messages from itself and its leader. In case 1, some process has sent a DECIDE message in round k, and hence, has decided in a lower round, which contradicts the definition of round k. We now consider case 2.

Suppose p_i decides x in round k. As p_i receives a majority of COMMIT messages in round k, and it decides on the estimate of one of the COMMIT messages (namely, the one from itself). From Lemma 7, all the COMMIT messages include the same estimate x and the same leader, say p_l. Thus p_i receives (k, COMMIT, x, $k-1$, p_l) from a majority of processes, and hence, a majority of process commits in round $k-1$ with estimate x. Let us denote this majority of processes by S_x.

We claim that if any process commits or decides in round $k' \geq k-1$, then it commits with estimate x or decides x. The claim immediately implies agreement. We prove the claim by induction on round number k'.

Base Case. $k' = k-1$. As processes in S_x commit x in round $k-1$, from Lemma 6, no process commits with an estimate different from x in round $k-1$. By definition of k, no process decides in round $k-1$.

Induction Hypothesis. If any process commits or decides in any round $k1$ such that $k-1 \leq k1 \leq k'$, then it commits with estimate x or decides x.

Induction Step. We need to show that if any process commits or decides in round $k'+1$, then it commits with estimate x or decides x. Suppose by contradiction that some process p_j commits with estimate $z \neq x$ in round $k'+1$. Then p_j has not received any DECIDE message in round $k'+1$. Also note that p_j commits on the estimate of the round $k'+1$ message that has the highest timestamp among all messages received by p_j in round $k'+1$. Let this highest timestamp be $tsMax$. Therefore, some process has sent a round $k'+1$ message with timestamp $tsMax$ and estimate z. From Lemma 8, some process commits in round $tsMax$ with estimate z.

Since the highest timestamp that can be received in round $k'+1$ is k', $tsMax \leq k'$. Since p_j commits in round $k'+1$, it has received round $k'+1$ messages from a majority of processes, and hence, received round $k'+1$ message from at least one process in S_x, say p_a. Recall that every process in S_x commits in round $k-1$ with estimate x. Thus, p_a has timestamp $k-1$ at the end of round $k-1$. As p_j has not received any DECIDE message in round $k'+1$, p_a has not decided by round k'. From Lemma 5, the round $k'+1$ message of p_a contains timestamp greater than or equal to $k-1$. Thus, $tsMax \geq k-1$.

Thus we have $k-1 \leq tsMax \leq k'$. By the induction hypothesis, every process that commits in round $tsMax$ commits $x \neq z$; a contradiction.

If some process p_b decides a value y in round $k'+1$, then in that round, either some process sends a DECIDE message with decision value y or p_b sends a COMMIT message with estimate y. By the induction hypothesis, $y = x$ in both cases. □

Lemma 10 *In every run r, all correct processes decide by round $GSR(r) + 2$.*

Proof : First, observe that in the eventually synchronous model, every correct process executes an infinite number of rounds, and in particular, executes round $GSR(r) + 2$.

We prove the lemma by contradiction. Assume that some correct process p_j does not decide by round $GSR(r) + 2$ in some run r. If any correct process p_i decides by round $GSR(r) + 1$, then it sends a DECIDE message in round $GSR(r) + 2$, and all correct processes receive that message and decide in round $GSR(r) + 2$, contradicting our assumption. Therefore, our assumption implies that, no correct processes decides by round $GSR(r) + 1$.

Let p_l be the correct process with the highest id in r. Since correct processes receive messages from all correct processes in all rounds from $GSR(r)$ onward, it follows that p_l is the leader of all correct processes in all rounds from $GSR(r)$ onward.

Consider round $GSR(r)$. We claim that at the end of round $GSR(r)$, no process has a higher timestamp than p_l. Suppose by contradiction that some other process p_j completes round $GSR(r)$ with a higher timestamp than p_l, say timestamp k'. There are three cases depending on when p_j adopts timestamp k': (1) p_j adopts timestamp k' before round $GSR(r)$, (2) p_j adopts timestamp k' on receiving a message from some process p_m in round $GSR(r)$ with timestamp k', or (3) p_j commits in round $GSR(r)$ and adopts $k' = GSR(r)$ as its timestamp. In the first two cases, since only correct processes enter round $GSR(r)$, and correct processes receive messages from all correct processes in round $GSR(r)$, p_l receives a message with timestamp k' (from p_j in the first case, and from p_m in the second case) and adopts a timestamp not smaller than k'; a contradiction.

Consider the third case. We show that p_l commits in round $GSR(r)$. In round $GSR(r)$, as correct processes receive message from all correct processes, every process evaluates $nextLD$ to p_l, and evaluates $maxTS$ to the same timestamp, say ts'. Since p_j commits in round $GSR(r)$, from condition commit-3, the leader of p_j in round $GSR(r)$ is same as its $nextLD$, i.e., p_l. From condition commit-2, it follows that p_j received a message ($GSR(r)$, $*$, $*$, ts', p_l) from p_l. Thus, p_l is its own leader at the beginning of round $GSR(r)$. Thus, at p_l, condition commit-3 holds. As all correct processes receive the same set of messages in round $GSR(r)$, and p_j and p_l have the same leader in round $GSR(r)$, commit-1 and commit-2 hold also at p_l. Thus, p_l commits in round $GSR(r)$, and hence, updates its timestamp to $GSR(r) = k'$; a contradiction.

Let ts'' be the timestamp of p_l at the end of round $GSR(r)$. Consider round $GSR(r) + 1$. Clearly, p_l sends ($GSR(r) + 1$, $*$, $*$, ts'', p_l). Every process on receiving this message evaluates $maxTS$ to ts''. At every correct process, p_l is the leader, and $nextLD$ is evaluated to p_l. Thus, all three conditions required to commit hold at every correct process. As no correct process decides by round $GSR(r)+1$, every correct process commits in round $GSR(r) + 1$. In the next round, every correct process sends the message ($GSR(r) + 2$, COMMIT, $*$, $*$, p_l). In round $GSR(r) + 2$, every correct process receives COMMIT messages from a majority that includes itself and p_l, and hence, decides; a contradiction. □

Lemma 11 *Algorithm UC_1 solves consensus.*

Proof : From Lemma 10, every correct process decides (termination). Validity holds since estimates are initialized to the proposal value and can only be set to other estimate values received in messages, and the decision value is one of the estimates. Uniform agreement is proven in Lemma 9. □

Theorem 2 *There is a consensus algorithm in eventually synchronous model with $t < n/2$ such that (a) in every run r, correct processes decide by round $GSR(r)+2$, and (b) in every nice run r', correct processes decide by round 2.*

Proof : From Lemma 11, UC_1 solves consensus. Part (a) follows from Lemma 10. To see part (b), consider any nice run r' of UC_1. In r', all processes are correct and receive messages from all processes in every round. Therefore, p_n is the leader at all processes in every round. In round 1, processes receive PREPARE messages from all processes with leader set to p_n and timestamp set to 0. So processes commit in round 1, and send COMMIT messages in round 2. On receiving COMMIT messages from all processes, processes decide in round 2. □

6 A Matching Algorithm for $t < n/3$

We now present a consensus algorithm UC_2 in the eventually synchronous model assuming $t < n/3$. The algorithm matches the lower bound of Theorem 1(a), and hence, also the lower bound of two rounds in nice runs. UC_2 is inspired by by algorithm A_{f+2} of [10], which in turn is inspired by [26]. However, A_{f+2} and [26] require that processes receive at least $n - t$ messages in every round, and therefore can have unbounded recovery times (see Section 2). We apply the timestamping scheme of UC_1 to obviate this requirement.

Algorithm UC_2 is presented in Figure 4. The algorithm is based on the following simple observation. Suppose $t < n/3$, and S is a multiset of n elements where some element v appears $n - t$ times. Then in any multiset containing $n - t$ elements from S, v appears at least $n-2t$ times and all other elements of S appear less than $n - 2t$ times.

We assume that some order is defined on proposal values. In every round, each process p_i sends its three primary

at process p_i

1: $k_i \leftarrow 0$; initialize()
2: **while** true **do**
3: $k_i \leftarrow k_i + 1$
4: **for** j = 1 to n **do:** send round k_i message to p_j
5: receive messages
6: compute()

7: **procedure initialize()**
8: $est_i \leftarrow prop_{p_i}$ {*read the proposal value*}
9: $ts_i \leftarrow 0$; $msgType_i \leftarrow$ PREPARE; $maxTS_i \leftarrow 0$; $msgSet_i \leftarrow \emptyset$
10: round 1 message $\leftarrow (1, msgType_i, est_i, ts_i)$

11: **procedure compute()**
12: **if** $dec_{p_i} = \perp$ **then**
13: **if** received $(k_i$, DECIDE, $est', ts')$ **then**
14: $est_i \leftarrow est'$; $ts_i \leftarrow ts'$; $dec_{p_i} \leftarrow est_i$; $msgType_i \leftarrow$ DECIDE {*decision*}
15: **else if** received at least $n - t$ round k_i messages in round k_i **then**
16: $ts_i \leftarrow k_i$
17: $msgSet_i \leftarrow$ set of $n - t$ round k_i messages received by p_i with lowest sender ids
18: $maxTS_i \leftarrow$ **Max**$\{ts |\ (k_i, *, *, ts) \in msgSet_i\}$
19: **if** every message in $msgSet_i$ has identical est (say est') and has $ts = k_i - 1$ **then**
20: $dec_{p_i} \leftarrow est'$; $msgType_i \leftarrow$ DECIDE {*decision*}
21: **else if** there are at least $n - 2t$ messages in $msgSet_i$ with identical est (say est'') **then**
22: $est_i \leftarrow est''$
23: **else**
24: $est_i \leftarrow$ **Max** $\{est |\ (k_i, *, est, maxTS_i) \in msgSet_i\}$
25: round $k_i + 1$ message $\leftarrow (k_i + 1, msgType_i, est_i, ts_i)$

Fig. 4 Algorithm UC_2.

variables to all processes: (1) the message type $msgType_i$ initialized to PREPARE, (2) an estimate est_i of the decision value, initialized to the proposal value (read from $prop_{p_i}$), and (3) the timestamp ts_i of the estimate value, initialized to 0. In the computation sub-round, p_i decides if it receives a DECIDE message. If p_i receives less than $n - t$ messages in round k then it does not update its variables in that round. If p_i receives at least $n - t$ messages then it updates its timestamp to the current round number k_i and updates other variables as follows. First, it arranges all messages received in the round in ascending order of their sender ids, selects the first $n - t$ messages, and puts them in set $msgSet_i$. If every message in $msgSet_i$ has the same estimate, say est', and every message in $msgSet_i$ has timestamp $k_i - 1$, then p_i decides est'. If at least $n - 2t$ messages in $msgSet_i$ have the same estimate, say est'', then p_i adopts est''. Otherwise, among the estimates received with maximum timestamp, p_i adopts the maximum one (i.e., the order on proposal values is used in order to break ties). We now prove correctness of UC_2.

Lemma 12 *In every run r, all correct processes decide by round $GSR(r) + 1$.*

Proof : We prove the lemma by contradiction. Assume that some correct process p_j does not decide by round $GSR(r) + 1$ in run r. If any correct process p_i decides by round $GSR(r)$, then it sends a DECIDE message in round $GSR(r) + 1$, and all correct processes receive that message and decide in round $GSR(r) + 1$; contradicting our assumption. Therefore, our assumption implies that no correct processes decides by round $GSR(r)$.

Consider round $GSR(r)$. Recall that only correct processes enter the round, and all correct processes receive messages from all correct processes. It follows that every correct process receives at least $n - t$ messages, and receives the same set of messages. Since no correct process decides in that round, correct processes update their timestamp to $GSR(r)$, and compute identical $msgSet$s. Then, either every correct process receives some estimate at least $n - 2t$ times and adopts that estimate, or all processes adopt the maximum estimate with the maximum timestamp. In either case, since processes have identical $msgSet$s, they update their estimates to the same value. Thus, in round $GSR(r) + 1$, processes receive identical estimates from all correct processes with timestamp $GSR(r)$, and decide; a contradiction. □

Lemma 13 (Uniform Agreement) *No two processes decide differently.*

Proof : If no process ever decides then the lemma trivially holds. Suppose some process decides. Let k be the lowest round in which some process decides; say p_i decides in round k. Process p_i can decide either (1) by receiving a DECIDE message, or (2) by receiving PREPARE messages from $n-t$ processes with identical estimate values and with timestamp $k - 1$. In case 1, some process has sent a DECIDE message in round k, and hence, has decided in a lower round, which contradicts the definition of round k. We now consider case 2.

Suppose p_i decides x in round k. Then in round $k - 1$, at least $n - t$ processes update their timestamp to $k - 1$ and

their estimate to x. Let this set of at least $n-t$ processes be S_x.

We claim that if any process updates its estimate or decides in round $k' \geq k-1$, then it updates it estimate to x or decides x. This claim immediately implies agreement. We prove the claim by induction on round number k'.

Base Case. $k' = k-1$. From the definition of round k, no process decides in round $k-1$. Suppose some process p_j updates its estimate in round k. Then p_j has received at least $n-t$ messages. As $t < n/3$, at least $n-2t$ of those messages are from processes in S_x, and hence, contain estimate x, and less than $n-2t$ messages are from processes not in S_x. Thus p_j updates its estimate to x.

Induction Hypothesis. If any process updates its estimate or decides in any round $k1$ such that $k-1 \leq k1 \leq k'$, then it updates it estimate to x or decides x.

Induction Step. If any process updates its estimate or decides in round $k'+1$, then it updates it estimate to x or decides x. Suppose a process decides y in round $k'+1$. Then either (1) some process has decided y in a lower round and sent a DECIDE message in round $k'+1$, or (2) at least $n-t$ processes has updated their estimate to y in round k'. In the first case, from the induction hypothesis and our assumption that no process decides before round k, it follows that $y = x$. Consider the later case. Again from the induction hypothesis it follows that, by the end of round k', all processes in S_x has either decided x, retained their estimate x, or has crashed. As there are at least $n-t$ processes in S_x and two sets of size $n-t$ intersect, we have $y = x$.

Now suppose some process p_j updates its estimate in round $k'+1$. Then p_j has received at least $n-t$ messages in round $k'+1$. As $t < n/3$, at least $n-2t$ of those messages are from processes in S_x, and hence from the induction hypothesis, contain estimate or decision value x. Also, less than $n-2t$ messages are from processes not in S_x, and so, less than $n-2t$ messages can contain a value different from x. Thus p_j updates its estimate to x. □

Lemma 14 *Algorithm UC_2 solves consensus.*

Proof : From Lemma 12, every correct process decides (termination). Validity holds since estimates are initialized to the proposal value and can only be set to other estimate values received in messages, and the decision value is one of the estimates. Uniform agreement is proven in lemma 13. □

Theorem 3 *There is a consensus algorithm in eventually synchronous model with $t < n/3$ such that, in every run r, correct processes decide by round $GSR(r)+1$.*

Proof : Immediate from Lemmas 12 and 14. □

7 Conclusions

Our work was motivated by the observation that many distributed systems and algorithms implementing state machine replication operate in two modes: a fast normal mode in stable periods, and a slower recovery mode when recovering from unstable ones. Furthermore, we observed that in all existing algorithms, the performance difference between the two modes is substantial: in all previous algorithms we are aware of, recovery can take up to five rounds, which is three more than the optimal normal mode. We set out to explore whether the recovery mode is indeed inherently more costly than normal mode, and if yes, by how much. Not surprisingly, we have found that if $t \geq n/3$, there is an inherent price for recovery from failures and asynchrony. But somewhat surprisingly, we have shown that this penalty is only one round. Even more surprisingly, we have shown that if $t < n/3$, there is no cost to recovery, which can be as fast as the normal mode.

Our algorithms were given in the basic round model of [11]. We note that this model can be trivially simulated in a system where eventually message delays are bounded by a known constant D, local computation takes negligible time, and processes have access to synchronized clocks: in round k, a process sends round k messages at time $(k-1)D$ and delivers all round k messages that are received by time kD.

Acknowledgements We thank Yoram Moses, Petr Kouznetsov, and Bastian Pochon for helpful discussions.

References

1. ACM: Special issue on group communications systems. Communications of the ACM **39**(4) (1996)
2. Amir, Y., Tutu, C.: From total order to database replication. In: Proceedings of the 22th IEEE International Conference on Distributed Computing Systems (ICDCS-22) (2002)
3. Birman, K., van Renessee, R.: Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press (1993)
4. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
5. Charron-Bost, B., Schiper, A.: Uniform consensus is harder than consensus. Journal of Algorithms **51**(1), 15–37 (2004)
6. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: A comprehensive study. ACM Computing Surveys **33**(4), 1–43 (2001)
7. Cristian, F., Fetzer, C.: The timed asynchronous distributed system model. IEEE Transactions on Parallel and Distributed Systems **10**(6) (1999)
8. Dolev, D., Reischuk, R., Strong, R.: Early stopping in byzantine agreement. J. ACM **37**(4), 720–741 (1990)
9. Dutta, P., Guerraoui, R.: Fast indulgent consensus with zero degradation. In: Proceedings of the Fourth European Dependable Computing Conference (EDCC-4). Toulouse, France (2002)
10. Dutta, P., Guerraoui, R.: The inherent price of indulgence. Distributed Computing **18**(1), 85–98 (2005). A preliminary version appeared in the Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC-21), 2002.
11. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
12. El Abbadi, A., Skeen, D., Cristian, F.: An efficient fault-tolerant protocol for replicated data management. In: Proceedings of the 4th ACM Conference on Principles of Database Systems (1985)
13. Fischer, M.J., Lynch, N.A.: A lower bound for the time to assure interactive consistency. Information Processing Letters **14**(4), 183–186 (1982)

14. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
15. Friedman, R., Vaysburd, A.: Fast replicated state machines over partitionable networks. In: Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems (SRDS-16), pp. 130–137. IEEE Computer Society (1997)
16. Gafni, E.: Round-by-round fault detectors: Unifying synchrony and asynchrony. In: Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC-17), pp. 143–152. Puerto Vallarta, Mexico (1998)
17. Guerraoui, R.: Revisiting the relationship between non blocking atomic commitment and consensus problems. In: Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG-9) (1995)
18. Keidar, I., Dolev, D.: Efficient message ordering in dynamic networks. In: Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC-15), pp. 68–76. New-York, NY (1996)
19. Keidar, I., Rajsbaum, S.: On the cost of fault-tolerant consensus when there are no faults – a tutorial. Tech. Rep. MIT-LCS-TR-821, MIT (2001). PODC 2002 Tutorial
20. Keidar, I., Rajsbaum, S.: A simple proof of the uniform consensus synchronous lower bound. Information Processing Letters **85**(1), 47–52 (2003)
21. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
22. Lamport, L.: The part-time parliament. Tech. Rep. 49, Systems Research Center, Digital Equipment Corp, Palo Alto (1989). A revised version of the paper also appeared in ACM Transaction on Computer Systems, 16(2):133-169, May 1998
23. Lamport, L., Fischer, M.: Byzantine generals and transaction commit protocols. Technical Report 62, SRI International (1982)
24. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**(3), 382–401 (1982)
25. Lampson, B.: How to build a highly available system using consensus. In: Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG-10), pp. 1–15. Bologna, Italy (1996)
26. Mostefaoui, A., Raynal, M.: Leader-based consensus. Parallel Processing Letters **11**(1), 95–107 (2001)
27. Oki, B., Liskov, B.: Viewstamped replication: A general primary copy method to support highly available distributed systems. In: Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC-7), pp. 8–17. Toronto, Ontario, Canada (1988)
28. Santoro, N., Widmayer, P.: Time is not a healer. In: 6th Annual Symp. Theor. Aspects of Computer Science, *LNCS*, vol. 349, pp. 304–313. Springer Verlag, Paderborn, Germany (1989)
29. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. **22**(4), 299–319 (1990)
30. Thekkath, C.A., Mann, T., Lee, E.K.: Frangipani: A scalable distributed file system. In: ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pp. 224–237 (1997)