The Overhead of Consensus Failure Recovery

Partha Dutta[1] Rachid Guerraoui[1] Idit Keidar[2]
(1) Distributed Programming Laboratory, EPFL, CH 1015, Switzerland
(2) Department of Electrical Engineering, The Technion, Haifa, 32000, Israel

Abstract

Many reliable distributed systems are consensus-based and typically operate under two modes: a fast *normal* mode in failure-free periods, and a slower *recovery* mode following failures. A lot of work has been devoted to optimizing the normal mode, but little has been devoted to optimizing the recovery mode. This work seeks to understand whether the recovery mode is inherently slower than the normal mode. We focus on consensus algorithms that tolerate arbitrary periods of asynchrony as well as the crash of a minority of processes, and study the time-complexity of their recovery mode in periods of synchrony.

We show that recovery induces an inherent overhead of one communication round. More precisely, we show that consensus algorithms that can tolerate arbitrary periods of asynchrony, require three rounds even in synchronous runs where all crashes are initial. This is in contrast to the well-known tight bound of two rounds on failure-free synchronous runs of such algorithms. We also give for the first time a consensus algorithm that is optimal in both the normal mode and the recovery mode of synchronous runs. Moreover, our algorithm recovers from arbitrary periods of asynchrony significantly faster than any other consensus algorithm we are familiar with. The algorithm uses the weakest failure detector for consensus, Ω.

Key words: Consensus, crash failures, time-complexity, asynchrony.

1 Introduction

1.1 Background and motivation

State machine replication [22, 32] 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 [26] is often employed, where an instance of consensus is triggered for each user request (or transaction) or group of user requests [23, 5].

It is well-known that consensus is not solvable in an asynchronous system even if only one process can crash [15]. On the other hand, it is often unrealistic to assume a completely

synchronous system where there are 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 systems are called *eventually synchronous* [12]. (Algorithms designed for such systems are called *indulgent* in [19].) Partially synchronous models [12, 8] and asynchronous models enriched with failure detectors [5] are frequently used to model eventually synchronous systems.

A run in an eventually synchronous system may begin with an unbounded *unstable* period (also called asynchronous period), during which failures may occur, no latency bounds are guaranteed to hold, and the output of oracle failure detectors can be arbitrary. However, each run eventually enters a *stable* period, in which some latency bounds or guarantees on failure detector output do hold, and during which no faulty process sends any message (i.e., faulty processes crash before sending messages in stable period).[1] In this paper, we model an eventually synchronous system as progressing in rounds; in a given run, the round in which the system becomes stable is called the Global Stabilization Round (GSR).[2] We denote this model by *ES*.

We are interested in determining time-complexity bounds for consensus algorithms in eventually synchronous systems. Obviously, we cannot bound the number of rounds needed to achieve consensus in asynchronous periods, as this would contradict the celebrated FLP result [15]; consensus algorithms might not terminate during asynchronous periods. We can, however, bound the number of rounds needed to reach consensus in stable periods. Such a bound is useful because, in practice, systems are indeed failure-free and synchronous most of the time [8].

Different consensus algorithms make different assumptions about the system's behavior during stable periods. Some assume that the system is fully synchronous in stable periods, that is, all messages arrive within known bounds [12, 8]. Others, e.g., those based on a group membership mechanism [7], assume an *eventually perfect failure detector*, $\diamond P$ [5]. This failure detector outputs a list of suspected processes at each process, so that eventually correct processes do not suspect any correct process and suspect all faulty processes. $\diamond P$ can be constructed in model ES as follows: a process suspects every process from which it did not receive a message in the previous round: it is easy to see that one round after GSR, no correct process suspects another correct process, and every correct process suspects all faulty processes. Other algorithms use strictly weaker failure detectors, such as the *eventual leader*, Ω, which has been shown to be the weakest for consensus [4]. Failure detector Ω chooses a *leader* at each process, so that eventually all correct processes choose the same correct leader forever.

We consider applications that invoke a sequence of consensus instances (e.g., state machine replication). We define a *recovery instance* to be a consensus instance during which the system becomes stable, or if there is no such instance, the first instance invoked in a stable period that follows an unstable period or a process failure. All other instances in stable period are called *normal instances*. For example, in a synchronous run with initial failures,[3]

[1] We also say that a period is *synchronous* if all latency bounds holds in that period, but failures might occur.

[2] Our definition of GSR resembles the notion of Global Stabilization Time (GST) in [12].

[3] We say that p_i is initially failed in a consensus instance if p_i does not take any steps in that instance; p_i

the first consensus instance is a recovery instance, and all subsequent instances are normal instances. Observe that, in a synchronous run with no failures, all consensus instances are normal instances.

Algorithms in eventually synchronous systems are typically optimized to work fast during stable periods: they run a fast *normal* mode in normal instances, and a slower *recovery* mode for recovery instances. This design pattern is employed in numerous replication schemes, including state machine replication schemes à la Paxos [23, 33]; transaction-based schemes such as [30, 13]; virtually synchronous group communication systems, where the group membership algorithm is run in recovery mode [3, 7, 1]; and also replication engines based on group communication [20, 16, 2].

Whereas it is common to focus on the optimality of the normal mode in algorithms for eventually synchronous systems, the optimality of recovery mode is usually a neglected issue. Indeed, the recovery mode is typically much slower than the normal mode in all algorithms we are familiar with. To illustrate this, consider the Paxos algorithm [23]. In stable periods, the processing of an update request takes two rounds in normal mode, which has been shown to be optimal [21, 24]. But in recovery mode, it may take up to five rounds. More specifically, Paxos is a leader-based algorithm: as long as there is an elected leader that does not fail, the leader can have each consensus instance complete in two rounds. But when the leader fails, three additional rounds are needed in order to elect a new leader, which causes the recovery instance to take five rounds. Likewise, in group communication-based replication [20, 16, 2], the group membership algorithm is triggered whenever a failure occurs, which adds to the delay of the recovery instance.

A similar situation occurs following periods of asynchrony. Asynchrony may cause a correct leader to be (falsely) suspected in a leader-based scheme, or a group membership view to change in order to exclude (falsely) suspected members. Once the system becomes stable again, the algorithm executes in the slower recovery mode: in leader-based schemes, the recovery process re-elects a correct leader, and a group membership-based system triggers a membership algorithm in order in order to allow falsely suspected processes to re-join the group.

1.2 Contributions

Our goal is to understand whether the additional cost of failure recovery is inherent, and if yes, we would like to determine the minimal price of recovery. Can algorithms in eventually synchronous systems reach consensus[4] in 2 rounds in the recovery instance just like in the normal mode? Are 5 rounds as in Paxos required? Or is it some value in between? Furthermore, is there an algorithm that is optimal in both the normal and the recovery mode? The aim of this paper is precisely to address these questions.

In order to understand the inherent cost of recovery, we consider a single instance of any consensus algorithm in ES, and establish a 3 round lower bound on those runs of the algorithm in which the system is synchronous and all failures are initial. Note that the

is initially failed in a run if p_i does not take any steps in that run.

[4]In this paper, to measure the time-complexity of a consensus instance, we always consider the global decision, i.e., time required for all correct processes to decide (which we distinguish from a halt event).

consensus instance in such runs is a recovery instance. More precisely, we show that for every consensus algorithm in ES, where at least a third of the processes may fail, there is a synchronous run r such that processes only fail initially in r, and some correct process decides in round 3 or a higher round.

Thus we show in a precise way that the recovery instance is inherently more expensive than consensus instances that run in the normal mode. Moreover, our lower bound is proven for the model where the system is completely synchronous in stable periods (i.e., in ES), and hence applies to algorithms that use unreliable failure detectors, e.g., $\diamond P$ or Ω, which can be implemented in this model.

Next, we turn to construct an algorithm whose complexity matches our recovery lower bound. We assume that a minority of processes can fail by crashing, which is a resilience lower bound for consensus algorithms in eventually synchronous systems [12, 5, 19]. Like Paxos [23], our algorithm relies on a eventual leader election service provided by the Ω failure detector, which is the weakest failure detector for consensus [4]. Note that Ω is strictly weaker than the eventually perfect failure detector $\diamond P$ [5, 4], which in turn can be implemented in ES. Thus our algorithm works under weaker assumptions than needed in order to prove our lower bound. However, we evaluate the time-complexity of our algorithm by running it in ES, and measuring the running time of a recovery instance from the time when the first process enters GSR to the time when all correct processes decide, i.e., a global decision.

We construct our algorithm in two steps: In Section 5 we address the optimality of recovery from failures in synchronous runs, and in Section 6, we address recovery from asynchrony.

More precisely, in Section 5, we present a simplified version of our algorithm, which matches our failure recovery lower bound in synchronous runs. The algorithm recovers within 3 rounds from failures occurring any time in the course of a given synchronous run. In other words, consider a point t_0 in a synchronous run after which no failures occur, then all the processes decide at most 3 rounds after point t_0. Moreover, the algorithm takes 2 rounds in the normal mode. To the best of our knowledge, this is the first algorithm to achieve both lower bounds.

In Section 6, we add recovery from asynchronous periods into the mix. We extend the algorithm of Section 5 so that it recovers from periods of asynchrony substantially faster than any previous algorithm we are aware of. Specifically, once the system becomes stable, our extended algorithm reaches global decision in three rounds plus one message delay. Whether this additional message delay is necessary for recovery from asynchrony remains an open problem.

2 Related work

The time-complexity of consensus has been widely studied in the traditional synchronous model (denoted SCS) where t out of n processes may fail by crashing. In runs with $f \leq t-2$ failures, the tight bound is $f+1$ rounds for global decision [14, 25], and $f+2$ rounds for global halting [9]. If we consider a stronger variant of consensus, uniform consensus, where no two processes can decide differently, the corresponding tight bounds for global decision

and halting are both $f+2$ [21, 6]. In this paper, we only consider global decision to measure time-complexity.

In eventually synchronous systems (e.g., ES), any algorithm that solves consensus also solves uniform consensus [18]. Clearly, the $f+2$ round lower bound for uniform consensus in SCS extends to synchronous runs of consensus algorithms in ES. Furthermore, [11] has shown that the lower bound is tight for synchronous runs in ES. Thus the tight bound for synchronous runs with $f \leq t-2$ failures, are identical (for uniform consensus) in SCS and ES.

Let us now consider synchronous runs with initial failures only. A simple adaptation of the algorithm in [25] gives a uniform consensus algorithm in SCS that globally decides in two rounds in every run where the failures are only initial. However, our lower bounds shows that, in ES, if a third of the processes may fail, the global decision requires 3 rounds in synchronous runs with initial failures. Thus, our lower bound highlights an inherent difference in the time-complexity of uniform consensus algorithms in SCS and ES.

Even among consensus algorithms designed for ES, our lower bound points out a difference in time-complexity that depends on the number of processes that may fail. If *less than one-third* of the processes may fail in ES, translating one of the eventual leader-based algorithms in [29] to ES, gives us a consensus algorithm in ES that globally decides in 2 rounds in synchronous runs with only initial failures. On the other hand, our lower bound shows that, if *at least one-third* of the processes may fail, consensus algorithms require 3 rounds for similar runs.

To prove our lower bound, we slightly extend the traditional technique (initially devised in SCS) that builds a chain of runs such that, any consecutive pair of runs in the chain are indistinguishable to some deciding process [9, 27]. In our proof, we expand the set of runs being considered to include asynchronous runs which are indistinguishable from synchronous runs at some deciding process.

To our knowledge, our algorithm is the first algorithm in an eventually synchronous system that meets the 3-round lower bound for recovery mode, as well as the 2-round lower bound for failure-free runs. Previous algorithms based on the rotating coordinator approach, e.g., [28, 31], have achieved the 2-round lower bound for failure-free runs but, in runs with initial failures, these algorithms can take a number of rounds that is proportional to the number of failures. Leader-based algorithms like Paxos and the algorithm of [10] also achieve the 2-round lower bound in failure-free runs, and they recover from failures faster than rotating-coordinator algorithms. However, neither of these algorithms meets our recovery lower bound: there are synchronous runs with initial failures only in which [10] takes 4 rounds for global decision, and Paxos takes 5 rounds. One of the algorithms of [29] achieves our new 3-round lower bound, but does not achieve the 2-round failure-free lower bound.

Our algorithm also considers another important aspect of recovery: recovery from an asynchronous period. We investigate the time required for an instance of consensus to globally decide after some process enters GSR. In runs that are initially asynchronous, the algorithms of [28, 31, 29, 10] can execute an unbounded number of rounds after some process reaches GSR. This is because, 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, the processes lagging behind have to execute an unbounded number of rounds (and send and await an unbounded number of messages) in order to catch up.

The Paxos algorithm recovers faster than the aforementioned algorithms. However, if we define GSR such that, after GSR the system does not become perfectly synchronous, but there are no new failures and the output of Ω at all correct processes is the same correct process, then Paxos can take a time that is linear in the number of processes in the system to recover, as we now exemplify. In Paxos, before a new leader can complete a consensus instance, it needs to be elected with a unique *ballot number*. The leader's new ballot number has to be endorsed by a majority of the processes. If the leader hears a higher ballot number from another process, the leader aborts its current ballot and chooses a new, higher, ballot number. During asynchronous periods, it is possible that each process in the system will try to become a leader with a different ballot number. Consider a failure-free run in which each process has a different ballot number when the system stabilizes (i.e., when the Ω failure detector selects the same leader at all correct processes). Assume that in the first round after GSR, the leader hears from the $\lceil \frac{n+1}{2} \rceil$ processes that have the lowest $\lceil \frac{n+1}{2} \rceil$ ballot numbers. The leader then starts a new ballot with a number greater than all of these. In its second attempt, the leader hears from a different majority, which includes the process with the next lowest ballot number, and again aborts the ballot and chooses a bigger ballot number. This can continue $O(n)$ times, until the leader has heard from all the processes.

In contrast, the algorithm we present in Section 6 terminates within three rounds and one message delay after the first process enters GSR. We are not aware of any other algorithm that recovers from asynchrony so quickly.

3 Model and Problem Definition

3.1 The Consensus Problem

In the consensus problem, every process starts with an input value called the proposal value and eventually outputs a common decision value. Processes propose a value v through invoking primitive *propose*(v) and decide a value v through invoking primitive *decide*(v). A consensus algorithm must satisfy the following properties: (a) (*integrity*) no process decides twice, (b) (*validity*) if a process decides v then some process has proposed v, (c) (*agreement*) no two correct processes decide differently, and (d) (*termination*) every correct process eventually decides. Uniform consensus is a variant of consensus that differs only in the agreement property. It enforces *uniform agreement*: no two processes decide differently.

3.2 The Failure Model

We consider a distributed system consisting of $n \geq 3$ processes: $\Pi = \{p_1, p_2, ..., p_n\}$. In a given run of the system, at most a threshold t of the processes can fail by crashing. Processes that fail in the run are said to be *faulty*, all other processes are *correct*.

We consider deterministic algorithms. Every process executes the algorithm assigned to it, until it (possibly) crashes, i.e., there are no Byzantine failures.

For presentation simplicity, we make the following additional assumptions: a crashed process never recovers; and every pair of processes are connected by a *reliable* communication channel: the channel does not create, alter, or duplicate any message, and every message sent from a correct process to a correct process is eventually received. We discuss the applicability of our result to weaker models (where processes can recover and channels can be unreliable) in Section 7.

3.3 Communication Rounds

We consider a round-by-round model [17]. Computation proceeds in rounds with round numbers starting from 1 and increasing by 1 for each subsequent round. In each round, a process (a) sends messages to all processes, (b) receives some messages, and (c) updates its state depending on the messages received; and then proceeds to the next round. We say that a process p_i *suspects* process p_j in round k if, in round k, p_i does not receive the round k message sent by p_j. We say that a message m sent from p_j to p_i is *lost* in a run, if p_i never receives m in that run.

When we focus on the lower bound, we assume full-information algorithms. That is, in every round, processes exchange messages with all other processes, and every message sent by a process contains the entire state of the process.

We say that a run of a consensus algorithm 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.

3.4 Timing and Failure Detection

We consider *eventually synchronous systems*, which can be asynchronous for unbounded periods of time. If the system employs a distributed failure detector oracle, then the oracle's output at each process can be arbitrary for an unbounded prefix of each run. For the sake of proving the lower bound, we consider the strongest possible eventually synchronous system model: we assume that in every run there is a time after which the system becomes synchronous. We capture such systems through the following round-by-round model: in the *ES* model, the following properties holds in every run:

t-Resilience every process that completes any round k, receives round k messages from at least $n - t$ processes.

Eventual synchrony there is an unknown but finite round number GSR (Global Stabilization Round) so that every process that sends a message in round GSR or a higher round is a correct process, and in every round $k \geq GSR$, for every correct process p_i, no process completes round k without receiving the round k message from p_i.

Reliable messages for every round k, any message m sent from a correct process to a correct process at round k is eventually received (at round k or at a higher round). We say that m is *delayed* if m is received at a round later than k.

We say that a run in *ES* is *good* if $GSR = 1$ in that run. Note that in a good run, all faulty processes crash before sending any message in the run. In other words, good runs are synchronous runs in which all failures are initial: any process that fails, fails before sending the messages in the first round. *Nice runs* are good runs without any failure.

In order to strengthen our upper bound result, we present our algorithm (in Section 5) in the asynchronous round-by-round model enriched with the Ω failure detector, which is the weakest failure detector for solving minority-resilient consensus [4]. Failure detector Ω outputs a single process *leader* at each process, such that eventually the output at every correct process is the same correct process. Ω can be easily constructed in the *ES* model as follows: in the first round, $\Omega.leader$ is p_1, and in each round $k > 1$, $\Omega.leader$ at process p_i is the minimum process (id) from which p_i has received a round $k-1$ message at round $k-1$. Note that this simple implementation of Ω has the following properties:

- In nice runs of *ES*, all processes have the same leader (p_1) in all rounds.
- In good runs of *ES*, all correct processes have the same leader from the beginning of round 2 onward.
- In every run, all correct processes have the same leader from the beginning of round $GSR + 1$ onward.

4 The Lower Bound

Our lower bound states that, when $t \geq n/3$, every consensus algorithm in *ES* has a good run that requires at least three rounds for global decision. We consider here only binary consensus where the proposal values are restricted to 0 and 1. The lower bound immediately extends to consensus because every algorithm for consensus also solves binary consensus. Furthermore, we show the lower bound only for the case $n = 3$ and $t = 1$; generalizing the proof to the case $3 \leq n \leq 3t$ is a straightforward simulation (similar to the one in Theorem 6.27 of [27]).

Lemma 1 *Every consensus algorithm in* ES *with* $n = 3$ *and* $t = 1$ *has a good run in which some process decides at round* 3 *or at a higher round.*

Proof : Assume by contradiction that there exists a binary consensus algorithm A in *ES* with $n = 3$ and $t = 1$ such that, in all good runs of A, every process that decides, decides by the end of round 2. Recall than every consensus algorithm in ES also solves uniform consensus. Thus, algorithm A also satisfies uniform agreement: no two processes decide differently.

We consider four runs of A; only the first two runs are good runs. In each of the four runs, p_1 and p_3 propose 1 and p_2 proposes 0. The first two rounds of these runs are depicted in Figure 1. We now describe them in words.

- In the good run a, process p_1 crashes initially (i.e., before sending any messages). Without loss of generality, we can assume that p_2 and p_3 decide 0 at the end of round 2.[5]

- In the good run b, only process p_2 crashes initially. The decision value in b must be 1 because p_2 and p_3 cannot distinguish b from another good run b' in which all processes propose 1 and p_2 crashes initially. By consensus validity, processes must decide 1 in b'.

- Run c is constructed as follows. Only process p_3 crashes and it does so after completing round 2 and before sending any message in round 3. Process p_1 suspects process p_2 in round 1 (this is depicted by the absence of a message arrow from p_2 to p_1 in Figure 1(c)), and process p_3 in round 2. Process p_2 suspects p_1 in round 1 and p_3 in round 2. Process p_3 suspects p_1 in both round 1 and round 2. Processes p_1 and p_2 suspect p_3 in round 3 and higher rounds.[6] Process p_3 cannot distinguish the first two rounds of c from the first two rounds of a. To see why, notice that p_3 does not receive any message from p_1 in the first two rounds of both runs. Furthermore, p_2 distinguishes a from c only at the end of round 2, and hence, sends identical messages to p_3 in round 2 of both runs. Therefore, as in run a, p_3 decides 0 at the end of round 2 in c. Due to the uniform agreement property, p_1 and p_2 eventually decide 0 in run c.

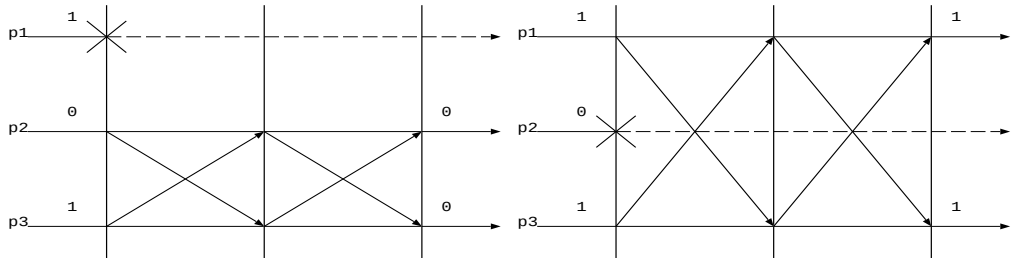
- Run d is constructed as follows. Only process p_3 crashes and it does so after completing round 2 and before sending any message in round 3. Rounds 1 and 2 are not synchronous. Process p_1 suspects process p_2 in round 1, and process p_3 in round 2. Process p_2 suspects p_1 in round 1, and p_3 in round 2. Process p_3 suspects p_2 in both round 1 and round 2. Processes p_1 and p_2 suspect p_3 in round 3 and higher rounds.[7] Notice that p_3 cannot distinguish the first two rounds of d from the first two rounds of b. Therefore p_3 decides 1 at the end of round 2. 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 1, the two runs differ only at process p_3 (because it receives different sets of message). Process p_1 receives the same messages in round 2 of runs c and d, and receives no message from p_3 in round 2 of both runs. Therefore, the state of p_1 is the same at the end of round 2 of runs c and d. Similarly, we can show that the state of p_2 is the same at the end of round 2 of runs c and d. Since process p_3 does not send any message after round 2, 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. □

[5] In case they decide 1, construct another run a', which is identical to a, except that p_1 proposes 0. As p_2 and p_3 cannot distinguish between a and a', they decide 1 in a'. In the subsequent runs, exchange the roles of p_2 and p_3.

[6] The delayed messages are handled as follows. Round 1 message from p_1 to p_2, and vice-versa, are delivered in round 2. Messages from p_1 to p_3 in round 1 and round 2, and messages from p_3 to p_1 and p_2 in round 2, are lost. Notice that the loss of these message does not violate the reliable messages property, as message delivery is guaranteed only between correct processes.

[7] The delayed messages are handled as follows. Round 1 message from p_1 to p_2, and vice-versa, are delivered in round 2. Messages from p_2 to p_3 in round 1 and round 2, and messages from p_3 to p_1 and p_2 in round 2, are lost.



(a) run a (b) run b

(c) run c (d) run d

Figure 1: Consensus runs of A

5 A Consensus Algorithm with Optimal Failure Recovery

We now present a new consensus algorithm, UC. The algorithm assumes that $t = \left\lfloor \frac{n-1}{2} \right\rfloor$, that is, a minority of processes may fail by crashing, and is presented in the asynchronous round-by-round model enriched with the Ω failure detector. In other words we assume a round-by-round model which has the *t-resilience* and *reliable messages* properties of ES, and at any point in the run, a process can query an Ω failure detector which returns a process id called *leader* such that: eventually, the query at every correct process returns the process id of same correct process.

We describe the algorithm in Section 5.1 and prove its correctness in Section 5.2. In Section 5.3 we show that if all correct processes have the same correct leader from round k onward, then UC ensures global decision by round $k+1$. Recall that, there is a simple implementation of Ω in ES, and hence, we can easily translate UC to ES. Furthermore, the output of our Ω implementation gives the same correct process at all correct processes, from the very beginning in nice runs, in round 2 in good runs, and in round $GSR+1$ in other runs. It follows that our algorithm globally decides in two rounds in nice runs, and in three rounds in good runs. Thus, the algorithm matches our lower bound for good runs, as well as the well-known lower bound for nice runs [21, 24]. This also implies that UC recovers from asynchrony in three rounds. However, following asynchronous periods, different processes may end up at different rounds, so in addition to the time needed for executing three rounds,

the algorithm also takes time to have all the processes reach round GSR. In Section 6 below, we will show how processes can quickly reach GSR once some correct process reaches GSR.

5.1 Algorithm Description

The UC algorithm is presented in Figure 2. It consists of two parallel tasks: Task 1 and Task 2. When a process proposes a value, it initiates both tasks. The algorithm terminates when one of the tasks returns.

Task 1. This task proceeds in Ω-based asynchronous rounds with processes moving incrementally from one round to the other. In each round, a process sends messages to all processes, waits for messages from a majority of processes including its current leader, computes its new state based on the messages received (possibly decides), and then moves on to the next round.

A process p_i maintains four local variables: the round number k_i, initialized to 1; an estimate of the decision value est_i, initialized to the proposal value; the current leader ld_i and the leader of the previous round $prevLd_i$, both initialized to $\perp$; and the type of the next message to be send $msgType$, initialized to A.

At the beginning of each round, p_i queries for the current leader and stores the identity in ld_i, the previous value of ld_i (i.e, leader at the beginning of the previous round) is stored in $prevLd_i$. We say that the *leader* of p_i is p_l in round k if p_i sends a message in round k with $ld_i = p_l$. A messages is sent with the following fields: k_i, $msgType_i$, est_i, and ld_i. We say that a process p_j is a *majority leader at round* k, denoted $majorityLd[k]$, if a majority of processes send round k message with $ld = p_j$. Since two majorities always overlap, there can be at most one majority leader at a given round. Note that in asynchronous periods, it is possible that there are rounds with no majority leader. A process p_i can send four types of messages in round k_i:

- If p_i has decided then it sends the decision value with $msgType = D$ message (lines 16 and 30), also called *decision message*.
- If p_i detects a majority leader p_l in some round $k_i - 1$ (i.e., receives message from a majority of processes with $ld = p_l$), receives message from p_l in round $k_i - 1$ (line 17 of round $k_i - 1$), and the leader of p_i is p_l in round $k_i - 1$ and k_i (line 8 of round k_i), then
  - if $p_i = p_l$ then p_i sends $msgType = C$ in round k_i.
  - if $p_i \neq p_l$ then p_i sends $msgType = B$ in round k_i.
- If the above conditions are not satisfied, then round k_i message of p_i is of type A.

Process p_i updates its est value at the end of a round k as follows:

- If p_i updates the $msgType$ to B then it also adopts the est value of the majority leader (line 21). If p_i updates the $msgType$ to C then p_i is the majority leader.

at process p_i

```
 1: procedure propose(v_i)
 2:   start Task 1; start Task 2
 3:   Task 1
 4:   k_i ← 1; est_i ← v_i; ld_i ← ⊥; prevLd_i ← ⊥; msgType_i ← A
 5:   while(true)
 6:     prevLd_i ← ld_i
 7:     ld_i ← Ω.leader
 8:     if (msgType_i ∈ {B, C}) and (ld_i ≠ prevLd_i) then
 9:       msgType_i ← A
10:     send(k_i, msgType_i, est_i, ld_i) to Π
11:     wait until (received(k_i, *, *, *) from ⌈(n+1)/2⌉ processes)
12:     if not received (k_i, *, *, *) from ld_i then
13:       wait until (received(k_i, *, *, *) from ld_i) or (ld_i ≠ Ω.leader)
14:     if (received (k_i, B, *, *) from ⌈(n+1)/2⌉ − 1 processes) and (received (k_i, C, *, *)) then
15:       est_i ← est of any round k_i message with msgType ∈ {B, C}
16:       decide(est_i); send (D, k_i + 1, est_i) to Π\p_i; return                    {decision}
17:     else if received(k_i, *, *, ld_i) from at least ⌈(n+1)/2⌉ processes including process ld_i then
18:       if p_i = ld_i then
19:         msgType_i ← C
20:       else
21:         msgType_i ← B; est_i ← est received from ld_i
22:     else if received (k_i, B, *, *) or (k_i, C, *, *) then
23:       est_i ← est of any round k_i message with msgType ∈ {B, C}
24:       msgType_i ← A
25:     else
26:       msgType_i ← A
27:     k_i ← k_i + 1
28:   Task 2
29:   upon receiving (D, k', x) do
30:     decide(x); send (D, k_i + 1, x) to Π\p_i; return                            {decision}
```

Figure 2: The consensus algorithm UC

- If the above condition is not satisfied (i.e., p_i does not update $msgType$ to B or C) and p_i receives some message with $msgType \in \{B, C\}$ then it adopts the est value contained in that message (line 23).
- Otherwise, the est value is kept unaltered.

A process p_i decides if it receives a majority of messages with $msgType \in \{B, C\}$ including at least one message with $msgType = C$ (line 14): the process decides on the est contained in any message with $msgType \in \{B, C\}$.

Task 2. Upon receiving a decision message ($msgType = D$) with value x, p_i forwards the decision value to other processes, and decides x.

5.2 Correctness of the Consensus Algorithm UC

Validity trivially holds, since the decision value is always an *est* value sent by some process, and *est* values are initiated to proposal values and can be changed only to other *est* values. We now prove Termination (Lemma 2) and Uniform Agreement (Lemma 4).

Lemma 2 (Termination) *Every correct process eventually decides.*

Proof : We prove the lemma by contradiction. Assume that some correct process never decides. If any correct process decides, then it sends a decision message to every process (lines 16 and 30). Since every message sent by a correct process to a correct process is eventually received, every correct process receives a decision message and decides, contradicting our original assumption. Therefore, our assumption implies that, no correct processes ever decides.

If some correct process p_i never decides, then either p_i is blocked forever in some round or executes an infinite number of rounds. We show that both cases are impossible.

Case 1. Some correct process is blocked forever in a round. Let k be the smallest round in which some correct process p_i is blocked forever. This is only possible at some **wait** statement.

Case 1.1. Process p_i is blocked forever at line 11 of round k. Since k is the smallest round in which some correct process is blocked forever, every correct process eventually completes line 10 of round k (i.e., sends round k messages to all processes). As every message sent by a correct process to a correct process is eventually received, and there are at least a majority of correct processes, p_i eventually receives round k messages from $\lceil \frac{n+1}{2} \rceil$ processes, and completes line 11.

Case 1.2. Process p_i is blocked forever at line 13 of round k. As in Case 1.1, every correct process sends round k messages to all processes. Consider the process that is ld_i at p_i at round k. There are two subcases. If ld_i is correct, then eventually, p_i receives the round k message from ld_i and completes line 13. If ld_i is faulty, then by the property of Ω, eventually $ld_i \neq \Omega.leader$. (Recall that, eventually, Ω outputs a correct process at all correct process.) Therefore, if ld_i is faulty then eventually, either p_i receives round k message from ld_i or $ld_i \neq \Omega.leader$. Process p_i completes line 13 in both cases.

Case 2. All correct processes execute an infinite number of rounds without deciding. From the property of Ω and the definition of faulty processes, we know that there is a time $t1$ after which, (1) only correct processes are up, and (2) Ω at every correct processes outputs the same correct process p'. Consider the lowest round k such that no process starts round k before time $t1$.

At round k, every correct process sets ld to p', and sends $(k, *, *, p')$ to all processes. Since no correct process ever decides, no correct process executes lines 15 and 16 (i.e., the condition in line 14 is evaluated to false). As there is a majority of correct processes and correct processes never suspect p' after time $t1$, then every correct process eventually receives $(k, *, *, p')$ from at least $\lceil \frac{n+1}{2} \rceil$ processes including process p'. Therefore, at round $k+1$, p' sends $(k+1, C, *, p')$ (since $ld = p'$) and other correct processes send $(k+1, B, *, p')$. Thus, at round $k+1$, every correct process receives at least one message of type C and at

least $\left\lceil \frac{n+1}{2} \right\rceil - 1$ messages of type B. Therefore, the condition of line 14 evaluates to true, and every correct process decides at line 16; a contradiction. □

Lemma 3 *For any round k, all round k messages with $msgType \in \{B, C\}$ have identical est values and identical ld values.*

Proof : Consider two distinct processes p and p' that send round k messages with $msgType \in \{B, C\}$. From the algorithm it is clear that $k > 1$ because every round 1 message is of type A (line 4). Furthermore, p and p' must have completed round $k-1$ with $msgType \in \{B, C\}$; i.e., at round $k-1$, both processes evaluated the condition in line 17 to true. Therefore, there exists a process p_l such that p received $(k-1, *, *, p_l)$ from a majority of processes. Similarly, there exist a process p'_l such that p' received $(k-1, *, *, p'_l)$ from a majority of processes. Since two majorities always overlap and every process sends at most one message at round $k-1$, we have $p_l = p'_l$.

From line 19, it follows that, if $p = p_l$ then p completes round $k-1$ with $msgType = C$ and its *est* unchanged. If $p \neq p_l$ then p completes round $k-1$ with $msgType = B$ and adopts the *est* send by p_l at that round (line 21). In either case, the *est* of p at the end of round $k-1$ is identical to the *est* of p_l at the beginning of round $k-1$. Similarly, *est* of p' at the end of round $k-1$ is identical to the *est* of p'_l at the beginning of round $k-1$. As we have already shown that $p_l = p'_l$, it follows that p and p' complete round $k-1$ with identical *est*. Thus, both processes send message at round k with identical *est* values.

To see that both processes send messages with identical *ld* at round k, notice that from the condition at line 17 of round $k-1$, p_l is the leader of p at round $k-1$ and p'_l is the leader of p' round at round $k-1$. Since $p_l = p'_l$, it follows that both processes have the same leader at round $k-1$. Furthermore, as both processes send messages with $msgType \in \{B, C\}$ at round k, they must have evaluated the condition at line 8 of round k to false; i.e., p has the same leader at rounds $k-1$ and k, and p' has the same leader at rounds $k-1$ and k. Thus, at round k, p and p' both have the same leader p_l, and send round k messages with $ld = p_l$. □

Lemma 4 (Uniform Agreement) *No two processes decide differently.*

Proof : If no process ever decides then the lemma trivially holds. Suppose some process decides. A process can decide either at line 16 of some round or at line 30 of Task 2.

We first show that if some process decides some values x at line 30, then some process has decided x at line 16. Suppose by contradiction that every process which decides x, decides at line 30. Consider the time $t1$ such that no process decides x at line 30 before time $t1$, and some process p_i decides x at line 30 at time $t1$. Thus p_i has received a message $(D, *, x)$ before time $t1$ at line 29. Since no process decides at line 30 before time $t1$, this must have been sent by some process at line 16. Thus some process decides x at line 16; a contradiction.

Consider the lowest round k at which some process p decides at line 16. Thus no decision message is sent at line 16 round k or at line 16 of a lower round. From the algorithm, p receives a majority of messages with $msgType \in \{B, C\}$, and p decides on the *est* of one of these messages. From Lemma 3, we know that all messages in round k with $msgType \in \{B, C\}$

have identical *est*, say d. Therefore, p decides d, every message with $msgType \in \{B, C\}$ has $est = d$, and there is a majority of such messages among the messages of round k.

We claim that *any process that decides at line 16 of round k, decides d, and every process that completes round k without deciding, does so with $est = d$.* This claim implies that *est* value of every process that completes round k is always d after round k. Since round k is the lowest round in which some process decides at line 16, it follows that no process decides a value different from d at line 16 of a round. Furthermore, we have already shown that if a process decides a value at line 30, then the same value has been decided by another process at line 16. Thus the above claim also implies that no process can decide a value different from d at line 30. Hence, uniform agreement follows from the above claim. Now, we prove the claim.

Suppose some process p', distinct from p, decides at line 16 of round k. From the algorithm, p' decides on the *est* value of some message with $msgType \in \{B, C\}$. However, every round k message with $msgType \in \{B, C\}$ has $est = d$. Therefore, p' decides d.

Now consider any process p'' that completes round k but does not decide at round k. Obviously, p'' does not receive any decision message at round k. (Otherwise, p_i decides at line 30.) We now show that p'' completes round k with $est = d$. There are two cases to consider:

Case 1. Process p'' evaluates the condition at line 17 of round k to false. In this case, p'' necessarily evaluates the condition at line 22 to true, because at round k there are a majority of messages with $msgType \in \{B, C\}$, and p'' receives a majority of round k messages (line 11). Therefore, p'' sets its *est* to *est* of some messages with $msgType \in \{B, C\}$. Recall that, every round k message with $msgType \in \{B, C\}$ has $est = d$. Hence, p'' completes round k with $est = d$.

Case 2. Process p'' evaluates the condition at line 17 of round k to true. Then, there exists a process p_l such that a majority of processes send round k messages with $ld = p_l$; i.e., $majorityLd[k] = p_l$. Furthermore, p'' completes round k with its *est* set to the *est* of p_l at the beginning of round k. (The $msgType$ of p'' is set to C if $p'' = p_l$, and $msgType$ is set to B otherwise.)

Consider the set of processes S that send round k messages with $msgType \in \{B, C\}$. From Lemma 3, every process in S sends a message with the same ld, say p'_l. Furthermore, p receives a majority of messages with $msgType \in \{B, C\}$ at round k, which implies that there is a majority of processes in S. Thus, $majorityLd[k] = p'_l$. Recall that, we already showed $majorityLd[k] = p_l$. Thus $p_l = p'_l$.

Notice that, from the condition at line 8 of round k, every process in S has the same leader at round k and $k-1$. Therefore, every process in set S sends a message in round $k-1$ with $ld = p_l$. Thus, $majorityLd[k-1] = p_l$. Furthermore, before p decides, it receives a $(k, C, *, *)$ message (line 14). Recall that any round k message with $msgType \in \{B, C\}$ has $est = d$. Therefore, p receives a message $(k, C, d, *)$, say from some process p_c. Thus, p_c sets $msgType$ to C at line 19 of round $k-1$, and the *est* of p_c at the beginning of round k is d.

Notice that a process can set its $msgType$ to C in some round, only if it is the $majorityLd$

in that round (lines 17 and 18). Therefore, $majorityLd[k-1] = p_c$. Recall that we have already shown that $majorityLd[k-1] = p_l$. Thus, $p_c = p_l$. Hence, the *est* of p_l at the beginning of round k is d. Recall that, p'' completes round k with its *est* set to the *est* of p_l at the beginning of round k. Thus, p'' completes round k with $est = d$. □

5.3 Performance of the Consensus Algorithm UC

We now discuss UC's time-complexity. The following lemma shows that once Ω outputs the same leader at all the correct processes, UC has all the processes decide in two rounds.

Lemma 5 *Suppose there exists a round k and a correct process p_c such that no Ω module at any process outputs a process id different from p_c at round k or at a higher round. Then, every process that decides, decides at round $k+1$ or at a lower round.*

Proof : Suppose by contradiction that some process p completes round $k+1$ without deciding. We first claim that p_c completes round k without deciding. If $p = p_c$ then we are done. Suppose $p \neq p_c$. From the definition of round k, at p, $\Omega.leader$ is p_c at round $k+1$. Since p completes round $k+1$ without deciding, from line 13, it follows that p receives a message m from p_c in round $k+1$. Message m is not a decision message, otherwise p decides in line 30 before completing round $k+1$. Thus, p_c does not send a decision message in round $k+1$, and hence, completes round k without deciding.

Consider any process p' that completes round k without deciding. Since p' does not decide at round k, it evaluates the condition at line 14 of round k to false (otherwise, it decides at line 16). From the definition of round k, every message sent at round k has $ld = p_c$. Furthermore, from line 13 it follows that p' receives round k message from p_c. Thus, p' sets $msgType$ to C if $p' = p_c$, and to B, otherwise.

As p completes round $k+1$ without deciding, in round $k+1$, p received round $k+1$ message from a majority of processes, none of which is a decision message. If a process p' sends a round $k+1$ message to p, and the message is not a decision message, then p' has completed round k without deciding. Thus, the round $k+1$ message from p' has $msgType = C$ if $p' = p_c$, and $msgType = B$ otherwise. Furthermore, from line 13 it follows that p receives round $k+1$ message from p_c. Thus, p evaluates the condition at line 14 of round $k+1$ to true, and hence decides at line 16 of round $k+1$; a contradiction. □

6 Speeding Up Recovery From Asynchronous Periods

In this section we consider the translation of the UC algorithm to ES.[8] Since from round $GSR+1$, the Ω implementation in ES returns the same correct processes at all correct processes, and all faulty processes crash in a lower round, from Lemma 5 it follows that, in UC, all correct processes decide by the end of round $GSR+2$. Unfortunately, this does not imply that UC recovers "quickly" from periods of asynchrony. This is because different processes may be in different rounds when some process enters GSR. If GSR=k, then it may

[8]We use the same name UC in model ES for the translated algorithm because the algorithm itself does not require any modifications for the translation; we only need to implement Ω in ES.

take $O(k)$ time for the processes that are behind to catch up and reach round GSR as well. In this section, we show how UC can be extended to recover quickly from asynchronous periods.

6.1 Description of the Recovery Booster

We now describe how to modify UC so that processes that are behind will be able to join the round of the faster processes within one communication delay from the time when the first process enters GSR.

We modify the wait statements of lines 11 and 13 so that p_i stops waiting once it receives a message $(k, type, e, l)$ with $k > k_i$ and $type \neq D$. Upon receiving such a message, p_i adopts all the values in the message, and changes the message type from B to C or vice versa according to whether p_i is the leader. That is, upon receiving a higher round message $(k, type, e, l)$ at line 11 or 13 with $type \neq D$, p_i executes the following:

$k_i \leftarrow k$; $est_i \leftarrow e$; $ld_i \leftarrow l$
if $(type \in \{B, C\})$ **then**
  if $(ld_i = p_i)$ **then** $msgType_i \leftarrow C$
  else $msgType_i \leftarrow B$
else $msgType_i \leftarrow A$
goto line 6

Now, p_i executes round k, and we say that p_i has skipped round k_i. The received round k message is not discarded: p_i counts it as one of the messages received in the wait condition of line 11 and line 13 of round k.

We call this modification of UC a *recovery booster*. Intuitively, modification to UC preserves correctness because the state received in a round k message from process p_j reflects a correct state reached at the end of executing $k-1$ rounds at p_j. This allows process p_i to "fast forward" through these rounds, and reach the same state that p_j reached.

6.2 Correctness of the Recovery Booster

In this section we show that UC augmented with the recovery booster suggested in Section 6.1 is correct. *Validity* still trivially holds. We now prove that Termination (Lemma 6) and Uniform Agreement (Lemma 8) also hold for UC with the recovery booster.

Lemma 6 (Termination) *Every correct process eventually decides.*

Proof : We need to show that the proof of Lemma 2 still holds despite the addition of the recovery booster. We revisit the two cases of the proof of Lemma 2, and show that both are still impossible with the recovery booster:

Case 1. Some correct process p is blocked forever in a **wait** statement in round k. If some correct process skips round k and sends a message in a later round $k' > k$, then p eventually receives this message and skips to round k', so p is not blocked in round k. But if no correct process skips round k, then the proof of case 1 in Lemma 2 holds.

Case 2. All correct processes execute an infinite number of rounds without deciding. Let $t1$ be a time after which (a) only correct processes are up, and (2) Ω at every correct processes outputs the same correct process p'. Consider, as in Lemma 2, the lowest round k' such that no process starts round k' before time $t1$.

We first prove that if a process skips round k', then there is a majority of processes that do send round k' messages, and moreover, p' sends a round k' message. A process can only skip a round k' if it receives a message from a later round sent by another process. Let p_i be the first process to send a message for any round $k'' > k'$. Then p_i cannot break from any **wait** statement in round k' using the recovery booster. Therefore, p_i completes round k' only after it receives round k' messages from a majority of the processes as well as from p' (since p_i's leader is p' from round k onward).

At round k', a majority of correct processes that includes p' set ld to p', and send (k', $*$, $*$, p') to all processes. Since no correct process ever decides, no correct process executes lines 15 and 16. Every correct process that does not break from the **wait** statement using the recovery booster, eventually receives (k', $*$, $*$, p') from at least $\left\lceil \frac{n+1}{2} \right\rceil$ processes including process p'. Consider the processes that terminate round k' without the recovery booster (there is at least one such process). If p' is one of these processes it sends ($k'+1$, C, $*$, p'), all other such processes send ($k'+1$, B, $*$, p'). A process that breaks from the **wait** statement upon receiving such a round $k'+1$ message sets its ld to p' and its $msgType$ to B or C, depending on whether it is p' or not. Since the leader at all processes remains p', the condition at line 8 is evaluated to false, and this process sends ($k'+1$, C, $*$, p') if it is p' and ($k'+1$, B, $*$, p') otherwise. Before any process can complete round $k'+1$ and move to round $k'+2$, it must receive messages from a majority that includes p'. Once this occurs, the process receives at least one message of type C and at least $\left\lceil \frac{n+1}{2} \right\rceil - 1$ messages of type B. Therefore, the condition of line 14 evaluates to true, and every correct process decides at line 16; a contradiction. □

We now show that the recovery booster does not violate safety. We first observe that Lemma 3 still holds:

Lemma 7 *For any round k, all round k messages with $msgType \in \{B, C\}$ have identical est values and identical ld values.*

Proof : Notice that a process that sets the $msgType$ to a value $\in \{B, C\}$ and skips to round k, also adopts the est and ld values sent in a round k message. This est value does not change before sending the message. The ld value can change in line 7, but if it does, the condition in line 8 evaluates to true and the $msgType$ becomes A. The proof follows from Lemma 3. □

Lemma 8 (Uniform Agreement) *No two processes decide differently.*

Proof : As in Lemma 4, we consider round k in which a process p receives a majority of messages with $msgType \in \{B, C\}$, and p decides on the est of one of these messages. From Lemma 7, we know that all messages in round k with $msgType \in \{B, C\}$ have identical est, say d. Therefore, p decides d, every message with $msgType \in \{B, C\}$ has $est = d$, and there

is a majority of such messages among the messages of round k. As in Lemma 4, uniform agreement follows immediately from the claim that *any process that decides at line 16 of round k, decides d, and every process that completes round k, does so with $est = d$.* The only new issue to deal with is the recovery booster: we need to show that processes that skip round k also do so with $est = d$. Indeed, a process skips round k only if it gets a round $k' > k$ message, whereupon it adopts the *est* value in that message. As shown in Lemma 4, a process that completes round k without the recovery booster does so with $est = d$. From a simple induction, it follows that no message from a round higher than k has *est* different from d. □

7 Conclusions

We have investigated the cost of recovery in consensus-based distributed systems that tolerate a minority of process crash failures as well as arbitrary periods of asynchrony. We have shown that, in a precise sense, the recovery instance is more expensive than consensus instances ran in the fast failure-free mode. We have also presented an algorithm, UC, which is optimal in both the recovery mode and the fast mode: UC takes 2 rounds if the run is synchronous and without failures, and takes 3 rounds when the run is synchronous and all failures are initial. More generally, UC has all the processes decide at most 3 rounds after all failures stop in synchronous runs. Algorithm UC tolerates a minority of crash failures, and uses the weakest failure detector for solving consensus in this model.

We have also explored another important aspect of recovery: recovery from asynchrony. Specifically, we studied how long it takes for all processes to decide after the first process entered GSR. The best previous algorithm that we know of, Paxos [23], requires 5 rounds plus one message delay to globally decide from GSR. Moreover, if from GSR the system does not become perfectly synchronous, but all the correct processes select the same correct leader (as guaranteed by Ω), then global decision in Paxos can take a time that is linear in the number of processes. We gave an extension of UC that globally decides within 3 rounds plus one message delay of the time when the first process enters GSR, even if from GSR the system is not perfectly synchronous but all correct processes have the same correct leader. An interesting open problem is to determine whether the extra communication step in addition to the 3 rounds is indeed necessary.

It is straightforward to extend UC to deal with message loss and crash recovery (as long as processes have access to stable storage). Message loss can be masked using retransmissions. Moreover, when the recovery booster of Section 6 is employed, each process only needs to worry about retransmitting its latest message; it is safe for all earlier messages to be lost. Thus, overcoming message loss can be very efficient, since it does not require logging and re-sending the entire message history. Crash-recovery can be easily tolerated using stable storage.[9]

[9] One simple approach could be to store the primary variables at a process (e.g., k_i, est_i, ld_i, $prevLd_i$, and $msgType_i$) into the stable storage, before sending the message in every round.

Acknowledgments

We thank Yoram Moses for helpful discussions, which helped us materialize the recovery booster. We also thank Petr Kouznetsov and Bastian Pochon for many helpful discussions on lower bounds of agreement problems.

References

[1] ACM. Special issue on group communications systems. *Communications of the ACM*, 39(4), April 1996.

[2] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the 22th IEEE International Conference on Distributed Computing Systems (ICDCS-22)*, 2002.

[3] K. Birman and R. van Renessee. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1993.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[6] B. Charron-Bost and A. Schiper. Uniform consensus harder than consensus. DSC Technical Report 2000-28, Department of Communication Systems, Swiss Federal Institute of Technology, Lausanne, May 2000.

[7] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, December 2001.

[8] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), June 1999.

[9] D. Dolev, R. Reischuk, and R. Strong. Early stopping in byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.

[10] P. Dutta and R. Guerraoui. Fast indulgent consensus with zero degradation. In *Proceedings of the Fourth European Dependable Computing Conference (EDCC-4)*, Toulouse, France, October 2002.

[11] P. Dutta, R. Guerraoui, and B. Pochon. Tight bounds on early local decisions. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC-17)*, Sorrento, Italy, October 2003.

[12] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[13] A. El Abbadi, D. Skeen, and F. Cristian. An efficient fault-tolerant protocol for replicated data management. In *Proceedings of the 4th ACM Conference on Principles of Database Systems*, 1985.

[14] M. J. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

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

[16] R. Friedman and A. Vaysburd. Fast replicated state machines over partitionable networks. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems (SRDS-16)*, pages 130–137. IEEE Computer Society, October 1997.

[17] E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC-17)*, pages 143–152, Puerto Vallarta, Mexico, 1998.

[18] R. Guerraoui. Revisiting the relationship between non blocking atomic commitment and consensus problems. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG-9)*, number 791 in Lecture Notes in Computer Science, pages 87–100, Le Mont-St-Michel, France, September 1995. Springer-Verlag.

[19] R. Guerraoui. Indulgent algorithms. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC-19)*, pages 289–298, Portland, OR, July 2000.

[20] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC-15)*, pages 68–76, New-York, NY, May 1996.

[21] I. Keidar and S. Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85(1):47–52, December 2002.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[23] L. Lamport. The part-time parliament. Technical Report 49, Systems Research Center, Digital Equipment Corp, Palo Alto, September 1989. A revised version of the paper also appeared in ACM Transaction on Computer Systems, 16(2):133-169, May 1998.

[24] L. Lamport. Lower bounds on consensus. *Unpublished note*, March 2000.

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

[26] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[27] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[28] A. Mostefaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 49–63, Bratislava, Slavak Republic, September 1999.

[29] A. Mostefaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, March 2001.

[30] B. Oki and B. Liskov. 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)*, pages 8–17, Toronto, Ontario, Canada, August 1988.

[31] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[32] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[33] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 224–237, 1997.