Computing in the Presence of Timing Failures

Gadi Taubenfeld
The Interdisciplinary Center
P.O.Box 167, Herzliya 46150, Israel
tgadi@idc.ac.il

Abstract

Timing failures refer to a situation where the environment in which a system operates does not behave as expected regarding the timing assumptions, that is, the timing constraints are not met. In the immense body of work on the designing fault-tolerant systems, the type of failures that are usually considered are, process failures, link failures, messages loss and memory failures; and it is usually (implicitly) assumed that there are no timing failures. In this paper we investigate the ability to recover automatically from transient timing failures. We introduce and formally define the concept of algorithms that are resilient to timing failures, and demonstrate the importance of the new concept by presenting consensus and mutual exclusion algorithms, using atomic registers only, that are resilient to timing failures.

1 Introduction

1.1 Motivation

One of the advantages of designing algorithms for asynchronous systems is that no assumption is made about the relative speed of the participating processes. Thus, such algorithms are robust in the sense that they (obviously) also operate properly in any partially synchronous system. However, this generality comes at the cost of efficiency and sometimes even solvability. Algorithms for asynchronous systems are usually less efficient than similar algorithms designed for synchronous systems, and important problems (such as consensus, renaming, election) do no have deterministic solutions in the presence of process failures using atomic read/write registers without making timing assumptions.

On the other hand, timing-based systems (where assumptions are made about the relative speed of the participating processes) enable the design of efficient algorithms. Furthermore, such systems are stronger than asynchronous systems enabling to solve problems that are unsolvable in asynchronous systems. A disadvantage of timing-based algorithms is that they may fail to operate properly when the timing constraints are not met. We will use the term *timing failure*, to refer to a situation where the timing constraints of a given timing-based system are not met.

To overcome the drawbacks of asynchronous systems (also called, time-free systems) and timing-based systems (or partially synchronous systems), as discussed above, we consider algorithms that are resilient to timing failures. These are efficient algorithms that are designed for timing-based systems with the following additional feature: their *safety* properties are always guaranteed to hold even in the presence of timing failures, and their *liveness* properties are guaranteed to hold as soon as the timing constraints are met.[1]

The appeal of such *time-resilient* algorithms lies in the fact that when they are executed in an asynchronous system, they "lie in wait" for a short period of time during which certain timing constraints are met, and when this happens these algorithms take advantage of the situation and efficiently complete their mission. Furthermore, when they are execute in a timing-based system such algorithms are very efficient as they are optimized for such a timing-based system. Finally, these algorithms guarantee that in case of timing failures, no harm is done – as soon as the timing constraints are again met the algorithms will *automatically* resume their normal efficient operation.

In the sequel, we formally define the notion of algorithms that are resilient to timing failures, and demonstrate the importance of the new concept by present several algorithms that are resilient to timing failures.

1.2 Timing-based Systems

A timing-based system provides an interesting abstraction of the timing details of concurrent systems. We as-

[1] Safety properties (such as, mutual exclusion, or never deciding on conflicting values) stipulate that "nothing bad ever happens", while liveness properties (such as, eventually entering a critical section, eventually reaching an agreement, or termination) stipulate that "something good eventually happens" [26].

Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)
0-7695-2540-7/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

sume that there is a *known* upper bound on memory access time. This assumption is inherently different from the asynchronous model where no such bound exists. While such a system abstracts from implementation details, in various environments it is a better approximation of the real concurrent systems compared to the asynchronous model, and enables to obtain more efficient solutions. Furthermore, it is stronger than the asynchronous model enabling us to solve problems that are unsolvable in the asynchronous model. We assume that the basic atomic operations are reads and writes of atomic registers, however, we add the following timing assumption about the speed of the processes,

- *Timing assumption:* There is an upper bound δ_i on the time it takes for process i to execute a statement which involves a single access to the shared memory. We assume that $\Delta = \max\{\delta_i\}$. That is, Δ is a *known* upper bound on the time it takes for the slowest process to execute a statement which involves an access to the shared memory.

- *Explicit delay statement:* Each process can delay itself by executing the statement:

  delay(d), for some constant d.

  A delay statement *delay*(d) by a process p delays p for at least d time units before it can continue. Since it is assumed that the value of Δ is known, delay statements can refer directly to Δ, and a process can enforce every other (nonfaulty) process to take at least one step by executing the statement *delay*(Δ).

Next, we define the notion of a timing failure.

A timing failure: *A timing failure refers to a situation where the timing constraints of a timing-based system are not met. In particular, a timing failure occurs when it takes more than Δ time units for a process to execute a statement which involves an access to a shared memory location.*

When measuring time complexity, we assume that the statement *delay*(Δ) takes exactly Δ time units, and that each access (local or remote) to a shared memory location takes at most Δ time units.

We have defined Δ to be the *maximum* time it takes for the slowest process in the system to execute a statement which involves an access to the shared memory. In systems which allow processes to be preempted (as almost all modern computing systems do), Δ must include the time spent preempted between two statements, in the time to execute one of them. Furthermore, in determining the value of Δ, possible cache misses, page faults, and delays due to memory contention should be taken into account. Thus, the value of Δ in most cases must be very large, rendered timing-based algorithms impractical, especially if they use delay also in the absence of contention.

However, because the time-resilient algorithms we present in this paper ensure correctness even when timing assumptions are violated, we can consider strategies in which we estimate Δ optimistically. Let us define *optimistic*(Δ) to be a bound on the time it takes each process in the system to execute a statement which involves an access to the shared memory *most* of the times (or even "enough" times), but not necessarily *all* the times. For example, *optimistic*(Δ) would ignore possible delays resulting from page faults, memory contention and preemption. Thus, the value of *optimistic*(Δ) can be significantly smaller than Δ. As we consider only algorithms that are resilient to timing failures, from a practical point of view, such algorithms would work correctly and are likely to be more efficient and practical when Δ is replaced with *optimistic*(Δ). The exact value of *optimistic*(Δ) should be tuned for each individual machine architecture, and can also be changed over time if the algorithm fails to make sufficient progress.

1.3 Tolerating Timing Failures

One negative effect of timing failures is to decrease efficiency. We will require that in algorithms that are resilient to timing failures, some time after timing failures stop, called *convergence time*, the algorithm becomes as efficient as if timing failures have never occurred. We will use the term *convergence* to refer to the time it takes to reach a configuration starting from which the time complexity of the algorithm is as if timing failures have never occurred. The notion of convergence is more interesting in the context of long-lived algorithms such as mutual exclusion, than in the context of short-lived algorithms such as consensus.

Tolerating timing failures: *An algorithm is resilient to timing failures, w.r.t. time complexity ψ, if it satisfies the following three requirements,*

1. **Stabilization.** The algorithm guarantees that *after* a transient timing failure has terminated, all the (safety and liveness) properties that the algorithm should satisfy immediately hold, assuming no more failures occur in the future. Furthermore, all the safety properties are always guaranteed to hold even during timing failures.

2. **Efficiency.** In the absence of timing failures (that is, when the timing constraints are *always* met), the time complexity of the algorithm should be ψ.

3. **Convergence.** The algorithm *convergence time* is finite. A finite number of time units after all timing failures stop, the time complexity of the algorithm is again ψ.

IEEE COMPUTER SOCIETY

In all the algorithms presented in this paper, the time complexity ψ always equals $c \cdot \Delta$ time units for some small constant c. Thus, in order to simplify the presentation, from now on instead of saying that an algorithm is resilient to timing failures w.r.t. time complexity $c \cdot \Delta$, we will simply say that an algorithm is resilient to timing failures (and implicitly assume that it is resilient w.r.t. time complexity $c \cdot \Delta$).

Weaker definitions of stabilization in which it is not required that the safety properties are guaranteed to hold during timing failures, or in which it is required that the safety properties only eventually hold (not necessarily immediately) are interesting. Also, it is possible to consider variants of the above definition in which the requirement that the time complexity of the algorithm after it converges is exactly the same as in the absence of timing failures, is relaxed. Stronger definitions of convergence in which it is required that the convergence time is a constant, or a function of the number of processes, are also interesting.

We point out that algorithms that are correct for completely asynchronous systems are usually, but not always, resilient to timing failures w.r.t. *some* time complexity ψ. However, in such cases, the value of ψ would usually be large, and such asynchronous algorithms be less efficient than the corresponding timing-based algorithms when executed in timing-based systems.

1.4 Results

We introduce and formally define the concept of algorithms that are resilient to timing failures. This new concept provides a novel way of looking at timing-based algorithms when designing fault-tolerant systems which may experience timing failures.

We assume that the reader is familiar with the definitions of the consensus problem and the mutual exclusion problem. To demonstrate the importance of the new concept we present two algorithms, using atomic registers only, that are resilient to timing failures:

- A consensus algorithm that is resilient to timing failures. The algorithm is also *wait-free*, which means that as soon as the timing constraints are met, it is guaranteed that each process eventually terminates even if all the other processes crash. That is, the algorithm can tolerate any number of process crash failures. Furthermore, the algorithm is *fast*: in absence of contention, a process decides after a *constant* number of its own steps, regardless of timing failures.

  Using the consensus algorithm as a building block, it is easy to design few other efficient algorithms that are resilient to timing failures. These algorithms include wait-free leader election, wait-free n-renaming, a wait-free implementation of a test-and-set object from atomic registers, etc.

  Given the known result about the universality of consensus [24], it follows from the fact that there exists a wait-free implementation of a consensus algorithm that is resilient to timing failures using atomic registers, that it is possible to provide a wait-free implementation that is resilient to timing failures w.r.t. some time complexity ψ of any object (which has sequential specification) using atomic registers only.

- A mutual exclusion algorithm that is resilient to timing failures. The algorithm is very efficient – has only $O(\Delta)$ time complexity – when there are no timing failures (that is, when timing constraints are met), and behaves as one of the best known asynchronous mutual exclusion algorithm during timing failures. That is, the algorithm alternates between these two modes of operation depending on the behavior of the environment.

The first result demonstrates that although reaching consensus in the presence of a single process failure is impossible using atomic registers in systems that are *always* asynchronous [22, 28], reaching consensus is possible in systems that are most of the time (but not always) asynchronous. The second result demonstrates that for systems that are not always asynchronous one can design a mutual exclusion algorithm that is efficient when the system behave in a completely asynchronous way, and is much more efficient during times when some timing constraints are met. As already mentioned, in order to make time-resilient algorithms practical, Δ must be replaced with *optimistic*(Δ).

1.5 Related Work

Timing-based algorithms, which are not resilient to timing failures, were considered in several papers and are covered in [34] (Chapter 10). The first timing-based mutual exclusion algorithm is due to Fischer and is described in [27]. Lamport's paper also contains a fast timing-based mutual exclusion algorithm which works correctly only when some bound is assumed on the time needed to execute the critical section [27]. Fast timing-based algorithms for mutual exclusion and consensus appeared in [4, 5, 6, 30].

In [29], a timing-based mutual exclusion algorithm is presented, in which only the property of deadlock-freedom depends on the timing assumptions, and mutual exclusion is guaranteed even in presence of timing failures. The deadlock (livelock) that is possible in the algorithm of [29] when there are timing failures, can persist even after there are no more timing failures. In [25] the time complexity of timing-based mutual exclusion algorithms is considered when counting only remote memory reference and delay statements.

IEEE COMPUTER SOCIETY

Message-passing algorithms for partially synchronous systems were presented in various papers [1, 19, 21, 35]. In particular, in [35] a leader election algorithm is presented which is correct under any assumption about time in the system, but exhibit graceful degradation of performance when the timing conditions deteriorate.

In [3] algorithms for both mutual exclusion and consensus were presented assuming that a time bound on the memory access time exists but is *not* known. The consensus algorithm in [3] proceeds in rounds where in each round the timing-based consensus algorithm from [6] is executed with some estimate for Δ. If no decision is made in a round then larger estimate for Δ is used in the next round. Our consensus algorithm is constructed similarly but, unlike the algorithm from [3], it is resilient to timing failures w.r.t. time complexity $c \cdot \Delta$. It follows from the lower bound proved in [3] that, in a model where a time bound on the memory access time exists but is *not* a priori known, there is no consensus algorithm with time complexity of $c \cdot \Delta$.

A similar approach is taken in constructing the randomized consensus algorithm from [11] with the delays removed from the algorithm in [3]. Time-resilient algorithms are related to randomized algorithms, as both approaches guarantee safety but relay on good luck (i.e., on the behavior of the scheduler) for liveness (such as termination) and efficiency. The unknown delay model is also considered in [23].

The concept of self-stabilization is due to Dijkstra [17, 18], which has defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps." We notice that self-stabilizing algorithms are not necessarily resilient to timing failures, as they are not required to satisfy the safety properties at all times. For example, two processes may be in their critical sections at the same time as a results of a failure. For a comprehensive description of results on the topic of self-stabilization see [20].

In [22], it was proved that reaching consensus in the presence of a single process failure is impossible in an asynchronous message-passing system. In [28], a similar impossibility result was proved for an asynchronous system which supports only atomic read/write registers. Timing-based systems enable to solve consensus by (indirectly) imposing constraints on the scheduler (i.e., the adversary's control is restricted). Another way to restrict the scheduler is explored in [9, 10], where it is shown how to solve consensus assuming priority-based scheduling and quantum-based scheduling. Another related approach, which provides a mechanism to circumvent timing assumptions in an asynchronous system, is the use of (unreliable) failure detectors [14, 15, 32]. Several other related papers are mentioned later in the paper.

2 Consensus in the Presence of Timing Failures

We present a consensus algorithm which is resilient to timing failures, using atomic registers only. The algorithm is *wait-free*, as soon as there are no more timing failures, one process cannot prevent another process from reaching a decision, and thus the algorithm can tolerate arbitrary number of crash failures. Furthermore, the algorithm is *fast*, in absence of contention a process decides after a *constant* number of its own steps, even in the presence of timing failures.

2.1 The Algorithm

The algorithm proceeds in rounds. The notion of a *round* is used only for the sake of describing the algorithm. We do *not* assume a synchronous model of execution in which all the processes are always executing the same round, and where no process can move to the next round before all others have finished the previous round.

Each process has a preference for the decision value in each round; initially this preference is the input value of the process. In each round, processes execute a timing-based consensus algorithm using Δ, the known upper bound on memory access time. The timing-based algorithm used in each round avoids conflicting decisions even if there is a timing failure during the round. If no decision is made in a round then the processes advance to the next round, and try again to reach agreement. If there are no more timing failures starting at the beginning of round r, than agreement is reached (at the latest) by the end of round $r+1$.

Algorithm 1. CONSENSUS IN THE PRESENCE OF TIMING FAILURES: `program for` p_i `with input` in_i.

shared registers
  $x[1..\infty, 0..1]$ array of bits, initially all 0
  $y[1..\infty]$ array, ranges over $\{\perp, 0, 1\}$, initially all $\perp$
  $decide$ ranges over $\{\perp, 0, 1\}$, initially $\perp$
local registers
  r_i integer, initially 1
  v_i bit, initially in_i

```
1 while decide = ⊥ do
2     x[r_i, v_i] := 1
3     if y[r_i] = ⊥ then y[r_i] := v_i fi
4     if x[r_i, v̄_i] = 0   then decide := v_i
5                          else delay(Δ)
6                               v_i := y[r_i]
7                               r_i := r_i + 1 fi
8 od
9 decide(decide)
```


IEEE COMPUTER SOCIETY

The algorithm uses the following shared data structures: a (two dimensional) infinite array $x[1..\infty, 0..1]$ of bits, and an infinite array $y[1..\infty]$ where the possible values of each $y[i]$ are $\{\perp, 0, 1\}$. The decision value is written to the shared register *decide*. In addition, each process p_i has a local register v_i, containing its current preference and a local register r_i, containing its current round number.

In round r, process p_i first flags its preference v by writing 1 to $x[r, v]$. Then, the process reads $y[r]$, and writes its preference to $y[r]$, if $y[r]$ has still its initial value $\perp$. Process p_i then reads the flag for the other preference (denoted by $\bar{v}$). If $x[r, \bar{v}]$ is not set, then every process that reaches round r with the conflicting preference $\bar{v}$ will find $y[r]$ set to v. Consequently, process p_i can safely decide on v. Otherwise, it waits for Δ time units and then sets its preference for the next round by reading $y[r]$.

Two processes with conflicting preferences for round r will not resolve the conflict only if both of them find $y[r] = \perp$ first and, as a result of a timing failure, one of them proceeds and chooses its preference for the next round before the other one finishes the assignment to $y[r]$. However, when there are no timing failures, each process finishes the assignment within time Δ and all the processes will choose the same value, the value of $y[r]$, as their preference for the next round. We notice that if all processes in a round have the same preference, then a decision is reached in that round. These two observations ensure termination.

Theorem 2.1 (Properties of the algorithm)

- *The algorithm is an efficient timing-based consensus algorithm that is resilient to timing failures. In the absence of timing failures each process decides and terminates after at most $15 \cdot \Delta$ time units (the first two rounds).*
- *If there are no more timing failures starting at the beginning of round r, then each process decides and terminates (at the latest) by the end of round $r + 1$.* ($r = \text{maximum}(\{r_j \mid j \in \text{set of all process' identifiers}\})$.)
- *The algorithm is wait-free – it can tolerate any number of process crash failures.*
- *The algorithm is fast – in absence of contention, a process decides after taking 7 steps, regardless of timing failures, and with no need to execute a delay statement.*
- *The number of participating processes is (potentially) unbounded. In particular, there is no need to know the number of participants.*

We observe that infinitely many atomic registers are used. Also, the convergence time is finite but unbounded. It is an interesting open problem to decide whether there is a time-resilient consensus algorithm which uses only finitely many atomic registers and/or has bounded convergence time. Notice that such an algorithm exists when there is a known bound on the number of time units during which there are timing failures.

Using the consensus algorithm as a building block, it is easy to construct algorithms that are resilient to timing failures for the other problems that have no fault-tolerant solutions using atomic registers, in a completely asynchronous system such as election, set-consensus and renaming.

2.2 Correctness

We now consider the correctness proof of the algorithm. For lack of space we only state the theorems and omit their proofs.

Theorem 2.2 (validity) *If p_i decides on a value v then $in_j = v$ for some p_j.*

Theorem 2.3 (agreement) *No two processes decide on conflicting values.*

Theorem 2.4 (termination & wait-freedom) *If from some point on there are no more timing failures, then it is guaranteed that each nonfaulty process eventually decides and terminates, regardless whether the other processes are faulty or not.*

3 Mutual Exclusion in the Presence of Timing Failures

We present an efficient mutual exclusion algorithm that is resilient to timing failures. The algorithm has only $O(\Delta)$ time complexity when there are no timing failures and behaves as one of the best known asynchronous mutual exclusion algorithm during timing failures. That is, the algorithm alternates between these two modes of operation depending on the behavior of the environment. Time complexity is defined as follows,

Time complexity: *The longest time interval where some process is in its entry code while no process is in its critical section, assuming there is an upper bound of Δ time units for step time in the entry or exit code and no lower bound.*

Notice that the above definition make sense for both timing-based and time-free algorithms. No known mutual exclusion algorithm achieves $O(\Delta)$ time complexity when executed in an asynchronous system. Moreover, there is no known mutual exclusion algorithm that works correctly in an asynchronous environment and at the same time has only $O(\Delta)$ time complexity when the time constraint of a timing-based environment are met. We present below the first such

IEEE COMPUTER SOCIETY

algorithm, which is constructed by combining two algorithms, a variant of Fischer's timing-based algorithm and an asynchronous fast starvation-free mutual exclusion algorithm.

3.1 Fischer's Timing-based Mutual Exclusion Algorithm

The first and most simple timing-based mutual exclusion algorithm is due to Fischer [27]. The algorithm uses one shared register named x. A process first waits until $x = 0$ and then assigns its *id* to x, delays itself and then checks x. If x has not been changed it can safely enter its critical section, otherwise it repeats this procedure. In the code of the algorithm, the statement "**await** *condition*" is used as an abbreviation for "**while** $\neg$*condition* **do** *skip*". The processes are numbered 1 through n.

Algorithm 2. FISCHER'S TIMING-BASED ALGORITHM: `process i's program.`

Shared x: atomic register, initially 0.

```
1  repeat   await (x = 0)
2           x := i
3           delay(Δ)
4  until    x = i
5  critical section
6  x := 0
```

The delay statement ensures that after a process finishes the delay statement, the value of x remains unchanged until some process leaving its critical section sets it to 0. In the absence of timing failures, Fischer's algorithm satisfies both mutual exclusion and deadlock freedom. However, it fails to guarantee mutual exclusion when there are timing failures.

3.2 A Space Lower Bound

It has shown by Burns and Lynch [13], that in an asynchronous system which supports only read/write atomic registers, any deadlock-free mutual exclusion algorithm for n processes must use at least n shared registers. Lynch and Shavit have shown that a similar lower bound holds also for timing-based systems in which the timing conditions are only *eventually* met [29]. This later lower bound implies a similar bound for time-resilient algorithms.

Theorem 3.1 *Any mutual exclusion algorithm for n processes that is resilient to timing failures w.r.t. some time complexity ψ must use at least n shared registers (regardless of the value of ψ).*

We observe that a mutual exclusion algorithm that is resilient to timing failures may during timing failures prevent all the processes from entering their critical sections. The algorithm presented below does not prevent all the processes from entering their critical sections during timing failures.

3.3 The Mutual Exclusion Algorithm

Let A be a correct mutual exclusion algorithms for asynchronous systems. That is A satisfies mutual exclusion and deadlock-freedom. The new algorithm, Algorithm 3, is obtained by replacing the critical section of Fischer's algorithm with a fast starvation-free mutual exclusion algorithm, called A, and replacing the single statement of the exit code of Fischer's algorithm with the statement: **if** $x = i$ **then** $x := 0$ **fi**. The code of Algorithm 3 is,

Algorithm 3. A MUTUAL EXCLUSION ALGORITHM: `process i's program.`

Shared x: atomic register, initially 0.
Assume the registers of A do not include x.

```
1  repeat   await (x = 0)
2           x := i
3           delay(Δ)
4  until    x = i
5  [ entry section of algorithm A ]
6  critical section
7  [ exit section of algorithm A ]
8  if x = i then x := 0 fi
```

The main question is: What should be the properties of Algorithm A which would guarantee that Algorithm 3 be resilient to timing failures?

First we observe that the fact that Algorithm A satisfies mutual exclusion for asynchronous systems implies that Algorithm 3 satisfies also mutual exclusion. It is easy to see that Algorithm 3 is deadlock-free, given that Algorithm A and Algorithm 2 are both deadlock-free.

In order for Algorithm 3 to satisfy the efficiency requirement in the definition of resiliency to timing failures we require that A be *fast*. That is, in absence of contention, a process must enter its critical section after a *constant* number of its own steps. Lamport's fast mutual exclusion algorithm [27] is the first published fast algorithm (using atomic registers). Lamport's algorithm satisfies deadlock-freedom but not starvation-freedom.

Several other fast algorithms, using atomic registers only, have been published [2, 7, 8, 12, 16, 31, 33]. All these fast algorithms satisfy starvation-freedom, however, they are rather complicated. A simple and elegant mutual

IEEE COMPUTER SOCIETY

exclusion algorithm (using atomic registers) which is both fast and starvation-free, can be easily obtained by applying a general transformation, due to Yoah Bar-David, which transforms any deadlock-free mutual exclusion algorithm into a corresponding starvation-free algorithm, to Lamport's fast mutual exclusion algorithm [27]. This simple transformation is described in [34], Problem 2.34. There are also simple fast starvation-free mutual exclusion algorithms which use synchronization primitives stronger than atomic registers.

Would using Lamport's original deadlock-free fast algorithm (for algorithm A) guarantee that Algorithm 3 satisfies also the third requirement (i.e.,convergence)? The answer is negative.

Theorem 3.2 *Let A be a fast deadlock-free mutual exclusion algorithm. Then, Algorithm 3 is not guaranteed to converge.*

Proof: As a result of time failures, several processes may execute the entry code of A at the same time. The fact that A is deadlock-free does *not* ensure that when there are no more timing failures, eventually at most one process will execute the entry code of A. Thus, because after a timing failures, there may always be contention in the entry code of A it may always take more than $c \cdot \Delta$ time units for a process to enter its critical section since a critical section was last released. Hence, the algorithm may never converge. ∎

Theorem 3.3 *Let A be a fast starvation-free mutual exclusion algorithm. Then, Algorithm 3 is guaranteed to converge. Thus, Algorithm 3 is resilient to timing failures.*

Proof: As a result of time failures, several processes may execute the entry code of A at the same time. The fact that A is starvation-free ensures that as soon as there are no more timing failures eventually at most one process will execute the entry code of A. To see that, we notice that a new process may reach the entry code of A only when another process sets x to zero in line 8. Line 8 guarantees that, out of all the processes that are concurrently in the entry code of A, at most one will set x to zero and all the others will simply exit leaving x unchanged. Thus eventually all the processes will leave the entry code of A.

We notice that as a result of a time failure several processes may be ready, at the same time, to set x to 0 in statement 8. This should cause no problem, as these processes are all guaranteed to finish executing statement 8 within Δ time units, after there are no more timing failures. Thus, when there are no more timing failures, after some process finishes executing statement 3 (the delay statement) all these processes will finish executing statement 8. ∎

To make Algorithm 3 practical, Δ must be replaced with *optimistic*(Δ). If Δ (or *optimistic*(Δ)) is *not* a priori known, we can start with a small estimated value and change it over time. One potential way to estimate Δ, is to use a technique similar to the one used in TCP congestion control (i.e, slow start and additive-increase, multiple-decrease).

4 Discussion

Real systems exhibit a significant degree of synchrony in practice, but few *guarantee* to do so. This synchrony can be exploited to achieve algorithms with properties that are not possible in a completely asynchronous systems, however, in general we cannot afford to risk incorrect behavior in case our assumptions about this synchrony are occasionally violated. We have explored the possible middle ground of exploiting synchrony when it is available, but in any case guaranteeing correctness regardless of the timing behavior of the system.

More precisely, we have investigated the ability to recover *automatically* from transient timing failures, and presented algorithms that are resilient to timing failures for the consensus and mutual exclusion problems. As we have pointed out, these algorithms would work correctly and be more efficient when Δ is replaced with *optimistic*(Δ). The exact value of *optimistic*(Δ) should be tuned for each individual machine architecture.

The notion of algorithms that are resilient to timing failure has been defined with respect to a pair of two models: the asynchronous model and a specific timing-based model. This notion can easily be generalized for any pair of two models A and B as follows, (1) a timing failure refers to a situation where the timing constraints of model A are not met, and (2) it is assumed that the timing constraints of model B are always met. Another possibility is to make assumptions directly about the scheduler. In this context, a *scheduling failure* refers to a situation where the constraints of the scheduler are not met. Resiliency in the presence of scheduling failures is defined in the obvious way.

There are numerous possible ways to extend this research: To solve other problems in various models as discussed above; to design efficient time-resilient concurrent data structures and local-spinning algorithms; to assume that both (transient) memory failures and timing failures are possible; to use synchronization primitives other that atomic registers; to use failure detectors; and to consider message passing systems.

Acknowledgement: I wish to thank an anonymous referee for many constructive suggestions.

References

[1] M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and con-

IEEE COMPUTER SOCIETY

sensus with limited link synchrony. In *Proc. 23rd ACM Symp. on Principles of Distributed Computing*, pages 328–337, 2004.

[2] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *Proc. 40th IEEE Symp. on Foundations of Computer Science*, pages 262–272, October 1999.

[3] R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. *SIAM Journal on Computing*, 26(2):539–556, April 1997.

[4] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, December 1992.

[5] R. Alur and G. Taubenfeld. How to share an object: A fast timing-based solution. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 470–477, December 1993.

[6] R. Alur and G. Taubenfeld. Fast timing-based algorithms. *Distributed Computing*, 10(1):1–10, 1996.

[7] J.H. Anderson and Y.-J. Kim. Fast and scalable mutual exclusion. In *Proceedings of the 13th international symposium on distributed computing*, 1999.

[8] J.H. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. *Proc. of the 14th international symp. on distributed computing. Lecture Notes in Computer Science*, 1914:29–43, 2000.

[9] J.H. Anderson and M. Moir. Wait-free synchronization in multiprogrammed systems: Integrating priority-based and quantum-based scheduling. In *Proc. 18th ACM Symp. on Principles of Distributed Computing*, pages 123–132, 1999.

[10] J.H. Anderson, M. Moir, and S. Ramamurthy. A simple proof technique for priority-scheduled systems. *Information Processing Letters*, 77(2-4):63–70, 2001.

[11] J. Aspnes. Fast deterministic consensus in a noisy environment. In *Proc. 19th ACM Symp. on Principles of Distributed Computing*, pages 299–308, July 2000.

[12] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. *Distributed Computing*, 15(3):177–189, 2002.

[13] J.N. Burns and N.A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

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

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

[16] M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[17] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[18] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5–6, 1986.

[19] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[20] S. Dolev. *Self-Stabilization*. The MIT Press, March 2000.

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

[22] 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, 1985.

[23] F. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-Free. In *19th international symposium on distributed computing*, September 2005.

[24] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[25] Y.-J. Kim and J.H. Anderson. Timing-based mutual exclusion with local spinning. In *17th international symposium on distributed computing*, October 2003. *LNCS 2848* Springer Verlag 2003, 30–44.

[26] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transaction on Software Engineering SE–3*, 2:125–143, 1977.

[27] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[28] M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[29] N.A. Lynch and N. Shavit. Timing-based mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 2–11, December 1992.

[30] M. M. Michael and M. Scott. Fast mutual exclusion, even with contention. Technical Report 460, Department of Computer Science, University of Rochester, June 1993.

[31] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. In *14th international symposium on distributed computing*, October 2000. *LNCS 1914* Springer Verlag 2000, 164–178.

[32] M. Raynal. A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News Distributed Computing Column 17*, 36(1), 2005.

[33] G. Taubenfeld. The black-white bakery algorithm. In *18th international symposium on distributed computing*, October 2004. *LNCS 3274* Springer Verlag 2004, 56–70.

[34] G. Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Education – Prentice-Hall, 2006. ISBN: 0131972596.

[35] P.M.B. Vitanyi. Distributed elections in an Archimedean ring of processors (preliminary draft). In *Proceedings of the 16th ACM Symposium on Theory of Computing*, pages 542–547, 1984.

IEEE COMPUTER SOCIETY