A Stabilizing Repair Timer

Ted Herman*

University of Iowa, Department of Computer Science
herman@cs.uiowa.edu

Abstract. Certain types of system faults, notably data errors due to transient faults, can be repaired by software. The repair consists of identifying faulty variables and then rewriting data to correct the fault. If fault identification is imprecise, repair procedures can contaminate non faulty processes from data originating at faulty processes. This contamination danger is resolved by delaying data correction for a sufficiently long period. In order to delay correction, processes use a repair timer. This paper considers the problem of how asynchronous processes can implement a repair timer that is itself subject to faults. The main results are requirement specifications for a distributed repair timer and a repair timer algorithm. The algorithm self-stabilizes in $O(\mathcal{D})$ rounds, where $\mathcal{D}$ is the diameter of the network, and provides reliable timing from k-faulty configurations within $O(k)$ rounds.

1 Introduction

Perfectly efficient repair requires precise knowledge of the minimal set of objects requiring some repair action. Several papers [7, 6, 12] have presented examples showing, for transient data faults, that a process cannot ascertain whether it is faulty itself or whether the fault is due to data corruption at a neighboring process. If the algorithm of a correct process trusts values from a faulty process, then the effect of even one initial fault could contaminate previously correct variables.

Some of the challenges of managing repair actions stem from conflicting priorities for fault tolerance. We would ideally like to have self-stabilization, which tolerates worst-case transient faults, and procedures that do not contaminate correct data during repair. Efficiency is also a consideration, so we strive for optimal stabilization times and the local sensitivity called time adaptivity in [12], where k process faults can be repaired in $O(f)$ time.

One way to overcome the danger of contamination is to use an optimistic or conditional approach to repair. New values for variables suspected of being faulty are computed, but not immediately installed. Instead, the repair procedure calculates tentative values and delays the installation of these values. The length of the delay could be given *a priori* by an optimistic bound related to the

* This work is supported by NSF CAREER award CCR-9733541.

suspected severity of a fault (using self-stabilization as a fall-back repair strategy), or the delay length could be dynamically set by procedures that verify the legitimacy of tentative values before they are installed.

Algorithms [9, 12] have used the tactic of delay to accommodate limited transient fault scenarios. Typical of this research is the classification of process variables into *output* variables, which are visible to protocol users, and internal variables. The term *output stabilization* is used in [12] to denote the final correction of output variables from an initially faulty state, whereas *state stabilization* refers to the correction of all variables. The delay tactic can be used to delay correction to output variables and can also be used to delay correction of certain internal "backup" variables. The latter is advantageous because output variables can be speculatively (or optimistically) corrected, leading to rapid output stabilization in certain cases, while the backup variables are corrected by a more conservative approach based on delay.

We propose the use of an asynchronous repair timer to provide adequate delay for optimistic repair layered upon self-stabilization. A repair timer can also be used to monitor and control optimistic repair; for instance if a repair procedure should complete in time $t \cdot f$ from an initial state with at most k faults, and if a fault condition persists after $t \cdot f$ time units, then a fall back repair is triggered.

Informally, the requirements of a repair timer are (i) it provides some means to count rounds (the asynchronous measure of time), (ii) it is time adaptive, providing each process a guaranteed number of "ticks" from any k-faulty initial state, (iii) it is efficiently self-stabilizing, and (iv) the timer provides interfaces so that processes can start the repair timer and inspect the elapsed time since repair initiation.

Measuring time in an asynchronous sense is the subject of many investigations, however few of these papers consider transient fault tolerance in an asynchronous system. Research on synchronizers, phase-synchronizers, and integer-valued clocks [2, 5, 8, 11, 13, 1] are the most closely related works to the repair timer of this paper. The only paper considering all cases of transient faults for a synchronizer in the asynchronous model is [3], but the algorithm presented there does not satisfy fault-locality properties in all cases. The algorithm presented in this paper follows the techniques of previous work, and is a small variation on the synchronizer theme. The contributions of this paper lie in the isolation of the requirements, the interfaces, and in a construction for the read/write atomicity model. As more researchers are investigating fault-local optimizations to self-stabilization, the explicit focus on repair timer abstractions is motivated.

2 System Model

The system consists of n processes communicating through link registers with read/write atomicity. The communication network is represented by an undirected, connected graph: for any pair of processes (p, q) there exists a pair of registers $(\mathsf{Register}_{pq}, \mathsf{Register}_{qp})$ if and only if there is an edge between p and q in the communication graph. If (p, q) is an edge in the graph, then we say p and q are neighbors. Process p is the only writer of $\mathsf{Register}_{pq}$ and q is the only reader of $\mathsf{Register}_{pq}$.

Let $\mathcal{D}$ denote the diameter of the communication graph. The distance between any pair (p, q) in the graph is denoted by $dist_{pq}$. To denote that p and q are neighbors, we use the notation $p \in \mathcal{N}_q$ and $q \in \mathcal{N}_p$. In the sequel, the term *region* refers to a connected component with a particular property.

A global state is given by the states of all registers and the local states of all process variables and program counters. A process step is either a register operation (and advancement of the program counter) or some modification of internal and output variables (and program counter) of that process. A computation is an infinite sequence of states so that each consecutive pair of states corresponds to a process step and the sequence of states includes an infinite number of steps of each process (computations are fair). The program of each process specifies a *cycle*, which consists of three parts: (i) a process reads one register written by each of its neighbors, (ii) the process assigns values to variables, and (iii) the process writes registers for each of its neighbors.

The system is designed to satisfy some task, and a global state σ is called *legitimate* if the task is satisfied in state σ and any process step from state σ results in a legitimate state. This definition of legitimacy is recursive, and invariants are used to characterize legitimate states of programs. The state predicate $\mathcal{L}$ characterizes legitimacy: state σ satisfies $\mathcal{L}$ iff σ is legitimate. Another state predicate $\mathcal{O}$ characterizes output legitimacy. A global state σ is called *output legitimate* iff $\mathcal{O}$ holds in state σ. If $\mathcal{O}$ is defined, then $\mathcal{L} \Rightarrow \mathcal{O}$, but the converse may not hold.

A state is k-faulty if it can be obtained from a legitimate state by modifying variables and write registers of at most k processes. For an k-faulty state, we call those processes faulty whose variables and/or write registers were modified to differ from their legitimate values. A *faulty region* in a state is a connected component of faulty processes.

Because each iteration of a process program specifies a cycle, time is conveniently measured in asynchronous rounds, which are defined inductively. A *round* of a computation, with respect to a start state σ, is a computation segment originating with σ of minimum length containing at least one complete cycle (from reading registers to writing registers) of each process. The first round of a computation consists of a round with respect to the initial state of the computation, and round k of a computation, $k > 1$, consists of a round with respect to the first state following round $k - 1$.

A system is *self stabilizing* if every computation contains a legitimate state (that is, for any initial state, the system eventually reaches a legitimate state). The *stabilization time* is the worst case number of rounds in any computation before a legitimate state occurs. The *output stabilization time* with respect to k faults is the worst case number of rounds in any computation starting from an k-faulty initial state before a suffix obtains so that $\mathcal{O}$ holds at each state in the suffix.

The repair timer will be added to a system that has known worst-case timing properties. From any initial state, the system stabilizes within $g \cdot \mathcal{D}$ rounds, where $g > 0$ is a known constant. For any k-faulty initial state, $k > 0$, the system satisfies two properties. First, for each faulty region X in the k-faulty state, some process p in X detects a faulty condition in the first round[1]. Secondly, the system output stabilizes within $h \cdot k$ rounds, where $h > 0$ is a known constant.

The assumption that self stabilization occurs within $O(\mathcal{D})$ rounds is based on works in the area of self stabilization showing that a large class of algorithms can be made self stabilizing with $O(\mathcal{D})$ stabilization times [4]. Different assumptions about system repair times, for instance $g \cdot \mathcal{D}^2$ stabilization time, would not drastically change the results of this paper. Even a system that is not self stabilizing can benefit from a repair timer. The assumptions of self stabilization, output stabilization, and constants g, h, are made to strengthen requirements given in Section 3 in the interest of generality. The construction presented in Section 4 is thus useful for systems that do not satisfy weaker assumptions, for instance, a system that does not self stabilize and only tolerates limited cases of k-faulty initial states.

3 Repair Timer Design

3.1 Repair Timer Interface

We consider the repair timer to be a subroutine called by each process in each cycle. Certain register fields are reserved for the repair timer; these register fields are read and written in each cycle so that the repair timers of different processes may communicate. The repair timer of each process has three method calls: `void start ( )` initiates the repair timer to start the timing of fault repair; `boolean doTick ( )` attempts to advance logical time for the repair, returning *true* iff the logical time advanced by the `doTick` call; and `int getTime ( )` returns the current "repair time" since the most recent `start` call.

A typical scenario for fault detection and repair for each process p using these methods is the following. In each cycle, p examines variables and register fields to check for a faulty condition. If p's examination reveals a faulty condition, p then calls `start` to ensure that the repair timer is active (note that p can observe a faulty condition for many successive cycles until the repair is finally complete; in such a case, p will call `start` in each of these successive cycles). After calling `start`, or if p does not call `start`, p invokes `doTick` to give the repair timer the opportunity to advance logical repair time.

If an invocation of `doTick` returns *true*, then that call is said to have generated a *tick*. Although p may not locally detect a faulty condition and invoke the `start` method, ticks can occur at p because fault repair has been initiated elsewhere in the system.

[1] For certain applications to contain the effects of corrupted data, it may be necessary to require that each non-faulty process neighboring X detect a faulty condition immediately so that erroneous values do not spread during computation.

3.2 Timing Properties

The repair clock provides ticks so that processes can monitor the progress of distributed repair procedures. Ticks must be related to rounds, which are the asynchronous unit of time measurement for computational progress (and repair). On one hand, a repair timer must produce sufficiently many ticks to be adequate for the worst-case repair times (specified by constants g and h in Section 2), but on the other hand the timer need not continue to produce ticks after repair is complete. The requirements given in this section formalize these two constraints. After the repair timer is added to a system, the resulting system should also be self-stabilizing. This entails that the repair timer itself be self-stabilizing. Note that the stabilization time of the repair timer can exceed the original system stabilization time of $g \cdot \mathcal{D}$ assumed in Section 2, so that adding a repair timer to a system may increase the system's stabilization time.

The essential requirement for a repair timer is that if process p observes ℓ ticks in some computation segment, one can infer that a process q at distance d from p has observed at least $\mathcal{C}(d, \ell)$ ticks, where $\mathcal{C}$ is some function locally computable by p. A process can thereby count ticks to determine the accuracy of values communicated by neighboring processes in the course of distributed repair. The following constraints for a function $\mathcal{C}$ form the basis for designing the repair timer. Let $\mathcal{C}$ be a function satisfying

$$\mathcal{C}(x+1, y) < \mathcal{C}(x, y) \quad \wedge \quad \mathcal{C}(x, y+1) > \mathcal{C}(x, y) \quad \wedge \quad \mathcal{C}(x, y) = \Theta(y)$$

These constraints state that $\mathcal{C}$ satisfies monotonicity properties with respect to its arguments and that $\mathcal{C}$ is a linear function with respect to the second argument. Assume that $\mathcal{C}$ is globally known, that is, $\mathcal{C}$ can be calculated by any process. For a fixed value $s \in [0, \mathcal{D}]$, define $\mathcal{C}_s(y) = \mathcal{C}(s, y)$ and let $\mathcal{C}_s^{-1}$ denote the inverse of $\mathcal{C}_s$ (which is well defined since $\mathcal{C}_s$ is a monotonically increasing function). Suppose also that $\mathcal{C}_s^{-1}$ can be calculated by any process.

Requirement 1 For any computation B starting from a k-faulty initial state, if t ticks occur at any process p, then for every q satisfying $dist_{pq} = d$, process q completed at least $\mathcal{C}(d, t)$ cycles in computation B.

Requirement 2 In any computation B starting from a k-faulty initial state, $k > 0$, each non-faulty process neighboring a faulty process observes at least $\mathcal{C}_k^{-1}(h \cdot k)$ ticks within $O(k)$ rounds.

Requirement 3 In any computation starting from a k-faulty initial state, $k > 0$, each process observes at least $\mathcal{C}_{\mathcal{D}}^{-1}(g \cdot \mathcal{D})$ ticks within $O(\mathcal{D})$ rounds.

Requirements 1–3 specify what is needed, at a minimum, for a repair timer. These requirements allow allow considerable latitude in an implementation. For example, many choices for a suitable function $\mathcal{C}$ may be possible; and Requirements 2–3 do not specify whether or not the repair timer should continue to generate ticks beyond some threshold. Inevitably, various design choices influence the form that a repair timer takes. For this paper, the design choice is to equip the repair timer with a clock_p variable, which enables each process p to monitor the progress of data repair (by the $\mathsf{getTime}$ method call mentioned in Section 3.1). Every tick returned by the repair timer corresponds to a change in the timer's clock variable.

Using a clock variable is a natural choice, given that counters are used in several distributed phase synchronization algorithms. Repair timers are differentiated from phase synchronization algorithms not only by the specifics of Requirements 1–3, but also by another design choice: the counter variables remain unchanged after repair completes (whereas in synchronization algorithms, counters continue to increment throughout computation). Define the constant $\mathcal{T}$ to be the "resting point" of the repair timer's counters.

Requirement 4 Every computation self-stabilizes to $(\forall p : \mathsf{clock}_p = \mathcal{T})$ within $O(\mathcal{D})$ rounds.

Part of the motivation for choosing a counter that eventually stops changing is an efficiency consideration. The repair timers of neighboring processes communicate the values of their clock variables via shared registers. Generally, most system computation lies outside a period of repair, and there is no need for activity by the repair timer during such times. If the clock values remain unchanged, then communication of clock values between processes can be greatly reduced (for instance, compressed to a single bit) for the value $\mathcal{T}$.

Requirement 4 suggests a convenient indicator of legitimacy for clock variables (e.g., the condition $\mathsf{clock}_p = \mathcal{T}$ can be tested in program statements). Fixing $\mathcal{T}$ as a resting point for the clock variables also makes it easy to start a sequence of ticks for the required timespan (see Requirement 2), and the start call amounts to assigning $\mathsf{clock}_p \leftarrow 0$.

Observe that if the system is in a legitimate state and a fault event occurs, then the state resulting from the fault event satisfies $\mathsf{clock}_p = \mathcal{T}$ for any non-faulty process p. This fact can be the key to fault containment and correct repair, since to contain faults it is enough that non-faulty processes refrain from changing variables until faulty processes complete their own repairs.

The decision to use clock variables to count ticks also suggests a useful alternative to Requirement 1. A process p can examine clock_p to determine the progress of fault repair. The following requirement presumes a known factor ξ_k, determined by the implementation of the repair timer, that satisfies $\xi_k = O(k)$ for an initially k-faulty state.

Requirement 5 In any computation B starting from a k-faulty initial state, if $\mathsf{clock}_p = t$ holds for a non-faulty process p, where $\xi_k < t < \mathcal{T}$, then for any process q such that $dist_{pq} = d$, process q completed at least $\mathcal{C}(d,t) - \xi_k$ cycles in computation B.

For Requirement 5 to be useful, ξ_k should be small enough so that the timings specified in Requirements 2 and 3 can be inferred from clock values. The construction in Section 4 provides an example.

4 Repair Timer Construction

Requirement 1 motivates a construction based on a synchronizer [2, 14]. The algorithmic observation of this paper is that Requirement 4 simplifies the synchronization task. Figure 1 shows code for the repair timer methods. For the code of Figure 1, let $\mathcal{T} \stackrel{\text{def}}{=} (g+3)\mathcal{D}$ and $\mathcal{C}(x,y) \stackrel{\text{def}}{=} y - 2x$. The latter definition satisfies the monotonicity and linearity constraints for $\mathcal{C}$, and the inverse $\mathcal{C}_x^{-1}(z) = z + 2x$ is simple to calculate. Also define ξ_k for a k-faulty initial state of a computation by $\xi_k \stackrel{\text{def}}{=} \min(k, \mathcal{D}) + 3$.

```
boolean doTick ( ):
  for  (q ∈ N_p)   x_p[q] ← read(Register_qp)
  tick_p ← false
  oldclock_p ← clock_p
  if  (∃q : q ∈ N_p : |clock_p − x_p[q]| > 1)  then clock_p ← 0
  else if  (clock_p < T ∧ (∀q : q ∈ N_p : x_p[q] ≥ clock_p))  then
              clock_p ← clock_p + 1
  if  (oldclock_p + 1) = clock_p  then  tick_p ← true
  for  (q ∈ N_p)  write(Register_pq ← clock_p)
  return tick_p

void start ( ):
  if  clock_p ≥ T − D  then  clock_p ← 0
  return
```

Fig. 1. repair timer methods for process p

The program of Figure 1 makes no provision for the case $\mathsf{clock}_p > \mathcal{T}$, as it is assumed that the domain of variable clock_p is $[0, \mathcal{T}]$ (the program can be easily enhanced to assign $\mathsf{clock}_p \leftarrow 0$ in case clock_p is found to be out of the intended range). The doTick method in Figure 1 shows **read** and **write** operations to emphasize that the doTick method is called in each process cycle. When these repair timer methods are integrated into a system, the **read** and **write** operations could be moved outside the doTick method.

Theorem 1. (**1**) The repair timer construction of Figure 1 satisfies Requirements 4 and 5; (**2**) For any computation in which some clock resets in the first three rounds, a state ρ occurs within $\mathcal{D} + 3$ rounds such that the computation beginning at state ρ satisfies Requirements 1 and 3; (**3**) For any computation starting at a k-faulty initial state such that some clock resets in the first three rounds, a state σ occurs within $3 + \min(\mathcal{D}, k)$ rounds such that the computation beginning at state σ satisfies Requirement 1 and satisfies Requirement 2 provided $k \leq \mathcal{D}$ and $g > 2h + 7$.

Proof. Section 5 is devoted to the proof of this theorem. Requirement 4 is proved by Lemma 8. Requirement 1 is addressed by Lemma 18. Lemma 19 verifies that

Requirements 1 and 3 are jointly satisfied within $\mathcal{D}+3$ rounds. Lemma 21 derives the stated conditions for which Requirement 2 holds. Requirement 5 is not explicitly proven by a lemma, but follows from the lemmas that address Requirements 2 and 3. □

The condition $g > 2h + 7$ is essentially a bound on $\mathcal{T}$, which needs to be large enough to guarantee sufficiently many ticks from a k-faulty initial state. The restriction $k < \mathcal{D}$ is for convenience in the derivation of Lemma 21, however it is also reasonable to propose $k < \mathcal{D}$ for optimistic repair, since Requirement 4 provides the fall back mechanism of self stabilization.

5 Verification

To prove Theorem 1, a sequence of lemmas build up invariants and time dependent properties of the construction in computations that start from a k-faulty initial state. To simplify the verification, most of this section considers the repair timer construction as an isolated system; the program for each system process is thus:

do forever: call doTick ( )

and a fault in this case is a change to variables (and/or program counter) in the doTick method of Figure 1. After proving Theorem 1 with this limited case of a fault and limited application of the repair timer, Section 5.4 considers integrating the repair timer into a system with repair procedures and fault detection procedures that invoke the start and doTick methods as set forth in Section 3.1. Due to space limitations, a number of proofs sketch arguments presented in [10].

Definition 1. A global state σ is a *final state* of the repair timer algorithm iff σ satisfies the weakest invariant F such that $F \Rightarrow (\forall p : \mathsf{clock}_p = \mathcal{T})$.

Definition 2. $\quad timediff_p \stackrel{\text{def}}{=} (\exists q :\ q \in \mathcal{N}_p :\ |\mathsf{clock}_p - \mathsf{clock}_q| > 1\ )$

Definition 3. A global state σ is a *smooth state* of the stop-clock algorithm iff σ satisfies the weakest invariant G such that $G \Rightarrow (\forall p : \neg timediff_p)$.

Definition 4. A global state σ is a *bumpy state* iff σ satisfies the weakest invariant H such that H implies

$$(\forall p, q :\ q \in \mathcal{N}_p\ \wedge\ |\mathsf{clock}_q - \mathsf{clock}_p| > 1 :\ \mathsf{clock}_p \in \{\mathcal{T}, 0\}\ \wedge\ \mathsf{clock}_q \in \{\mathcal{T}, 0\})$$

Bumpiness thus defines a predicate where the only exception to smoothness is a situation where the clock value $\mathcal{T}$ neighbors a clock value of zero.

Definition 5. A connected component B of the network is a *smooth region* iff

$$(\forall p, q :\ p \in B\ \wedge\ q \in \mathcal{N}_p :\ |\mathsf{clock}_p - \mathsf{clock}_q| \leq 1 \Rightarrow q \in B)$$

This definition partitions the set of processes into equivalence classes, where each equivalence class is a smooth region.

Definition 6. A *safe region* is a smooth region B satisfying

$$(\forall p, q : \ p \in B \ \wedge \ q \in \mathcal{N}_p : \ |\mathsf{clock}_p - \mathsf{clock}_q| > 1 \ \Rightarrow \ \mathsf{clock}_p = 0)$$

In a cycle, if process p executes $\mathsf{clock}_p \leftarrow 0$, then call that cycle a *clock reset cycle* of p. If process p changes its clock to a non zero value, then call that cycle a *clock increment cycle*.

5.1 Self-Stabilization

Lemma 1. Let p and q be any processes satisfying $q \in \mathcal{N}_p$, and let $\mathsf{x}_p[q]$ be p's internal copy of clock_q. After the second round of any computation, at each state,

$$\mathsf{clock}_q \in \{\mathsf{x}_p[q], \mathsf{x}_p[q]+1, 0, \mathsf{clock}_p, \mathsf{clock}_p + 1\} \ \vee \ \mathsf{clock}_p = 0$$

Proof. After one round in any computation, every process computes a value for its clock variable and writes that value into all its registers. After the second round, each x variable has therefore been assigned some value previously written from a clock variable. The conclusion can be shown following the second round by considering neighbors p and q and four cases for subsequent computation: (*1*) no step changes clock_p; (*2*) no step writes clock_p to $\mathsf{Register}_{pq}$, but $\mathsf{clock}_p \neq \mathsf{Register}_{pq}$ becomes true by assigning zero to clock_p; (*3*) no step writes clock_p to $\mathsf{Register}_{pq}$, but $\mathsf{clock}_p \neq \mathsf{Register}_{pq}$ becomes true by incrementing clock_p; or (*4*) clock_p increments and is written to $\mathsf{Register}_{pq}$. A straightforward analysis of these four cases verifies the lemma's conclusion. □

Lemma 1 can be used in simple arguments to prove the following pair of lemmas.

Lemma 2. In any computation, there exists a safe region after three rounds.

Lemma 3. In any computation B following round two and any consecutive pair of states (σ, σ') in B, if processes p and q belong to a common safe region at σ, then they belong to a common safe region at σ'.

Lemma 4. If p belongs to a safe region and $q \in \mathcal{N}_p$ does not belong to that region at some state following round two of a computation, then after one additional round, both p and q belong to a common safe region.

Proof. A simple argument by contradiction establishes the result. □

Lemma 5. In any computation, after at most $2 + \mathcal{D}$ rounds, a smooth state is obtained.

Proof. If all processes belong to a common safe region, then the state is smooth. A variant function can be constructed as an n-tuple, sorted in descending order, of the diameters of all safe regions in a given state (padding with zeros in the tuple if there are fewer than n safe regions). Tuples are lexicographically compared to define the variant ordering. Lemma 3 implies that regions do not shrink by any step; and Lemma 4 shows that the variant increases so long as the state is not smooth, since safe regions either grow in diameter or merge with other safe regions in each round, until the maximum of the variant function is obtained (one region of diameter $\mathcal{D}$). Finally, Lemma 2 shows that by round 3, the global state has at least one safe region. Therefore, after $3 + (\mathcal{D} - 1)$ rounds, the global state is smooth. $\square$

Lemma 6. For any computation B containing a smooth (bumpy) state σ in round three or higher, all states following σ in B are smooth (bumpy) states. [Smoothness and bumpiness are invariants.]

Proof. Invariance of smoothness and bumpiness can be verified by straightforward inspection of the code, based on Lemma 1. $\square$

Lemma 7. Starting from an initially smooth state, the clock algorithm stabilizes to a final state within $\mathcal{T}$ rounds.

Proof. Consider a smooth state σ for which the minimum clock value is $t < \mathcal{T}$. After one round the minimum clock value exceeds t; at most $\mathcal{T}$ increments to a clock are possible before reaching the value $\mathcal{T}$. $\square$

Lemma 8. The repair timer construction satisfies Requirement 4.

Proof. Lemma 5 establishes that each system computation contains a smooth state after at most $2 + \mathcal{D}$ rounds; therefore, using Lemma 7, it follows that each system computation stabilizes to a final state within $2 + \mathcal{D} + \mathcal{T}$ rounds. $\square$

5.2 Synchronization Properties

To simplify the statements of upcoming results, suppose in this subsection that all computations start with round three, that is, the conclusion of Lemma 1 holds throughout computation B in each of the following lemmas.

Lemma 9. Let B be a prefix of an arbitrary computation such that there exists a subsequence A of B and a value j for which the sequence of clock_p values in A is $j, (j+1), (j+2), \ldots, (j+\ell)$. Then, for $0 \le d < \ell/2$, for every process q satisfying $dist_{pq} = d$, there exists a subsequence A' of B and a value i for which the sequence of clock_q values in A' is $i, (i+1), (i+2), \ldots, (i+\ell-2d)$.

Proof. The proof is an induction on d. If $d = 0$, then $q = p$ and the lemma holds trivially by the choice $i = j$. By hypothesis, suppose the lemma holds for $d \le m$ and consider some process q such that $dist_{pq} = (m+1)$. Let r be a neighbor to q satisfying $dist_{pr} = m$. The only possible increase to clock_r is by a clock increment cycle, and a precondition of a clock increment cycle is that every clock value neighboring clock_r is equal or one greater than clock_r, which can be used to complete the induction. $\square$

Lemma 10. Let B be a prefix of an arbitrary computation such that there exists a subsequence of B and a value j for which the sequence of clock_p values is $j, (j+1), (j+2), \ldots, (j+\ell)$. Then B contains at least $\ell - 2d$ clock increment cycles for every process q satisfying $dist_{pq} = d$, $d < \ell/2$.

Proof. Lemma 9 shows that for each q at distance d from p, provided $d < \ell/2$, that q witnesses clock_q values of $i, (i+1), (i+2), \ldots, (i+\ell-2d)$. Thus q increases the value of clock_q at least $\ell-2d$ times in computation B. Since the only possible increase for a clock variable is a clock increment cycle, computation B has $\ell-2d$ clock increment cycles for process q. □

Lemma 11. For any computation B starting from a non-smooth initial state,

$$(\forall p : \ (\exists \sigma : \ \sigma \in B : \ \mathsf{clock}_p \leq \mathcal{D} \text{ at state } \sigma))$$

Proof. If the initial state of B is not smooth, then in the first round some process assigns its clock variable to zero. Lemma 7 guarantees that the repair timer algorithm stabilizes with every clock equal to $\mathcal{T}$. Let q be the *last* process to assign its clock variable zero in computation B, and let C be the suffix of B beginning after q's assignment $\mathsf{clock}_q \leftarrow 0$. An induction based on distance from q shows that any process s satisfying $dist_{qs} = d$ satisfies $\mathsf{clock}_s \leq d$ at some state in computation C. Since $d \leq \mathcal{D}$ by definition of diameter, the lemma holds. □

Lemma 12. In any computation starting from a non-smooth initial state, every process observes at least $\mathcal{T} - \mathcal{D}$ ticks.

Proof. Lemmas 11 and Lemma 7 imply the result. □

Lemma 13. Let B be a computation starting from a smooth initial state σ, and let t be the minimum clock value at state σ. Then for any process p with $s = \mathsf{clock}_p$ at σ, within ℓ rounds of computation B, process p observes at least $\min(\mathcal{T} - s, \ell - (s - t))$ ticks.

Proof. The proof is based on a variant function, using a potential res_p for a process p to increment its clock a certain number of times within ℓ rounds. At state σ, let $res_p = \min(\mathcal{T} - s, \ell - (s-t))$. For other states in B, res_p is defined by the following procedure: so long as $res_p > 0$, each increment to clock_p decrements res_p by 1. Also, define for each state $\delta \in B$ the function div_p to be the difference between clock_p and the minimum clock value in the system at state δ. The variant function for the proof is the tuple $\langle res_p\, , \ div_p \rangle$, and reduction in this variant is defined by lexicographic comparison of tuples.

Lemma 6 establishes the invariance of smoothness, and no clock variable decreases from a smooth state. In each round, the minimum clock value increments at least once, which implies that div_p decreases in a round for which $\mathsf{clock}_p \neq \mathcal{T}$ and clock_p does not increment. Therefore in each round, so long as $\mathsf{clock}_p \neq \mathcal{T}$, the variant $\langle res_p\, , \ div_p \rangle$ reduces. These ℓ reductions in the variant are due to increments of clock_p except for the initial difference $div_p = s - t$, present at state σ. That is, process p may increment its clock to increase div_p, but if $\ell > s - t$, then p must increment at least $\ell - (s - t)$ times (unless clock_p attains the value $\mathcal{T}$). □

Lemma 14. Let B be a computation starting from a bumpy initial state σ, and let t be the minimum clock value at state σ. Then for any process p with $s = \mathsf{clock}_p \neq \mathcal{T}$ at σ, within ℓ rounds of computation B, process p observes at least $\min(\mathcal{T} - s, \lfloor \ell/2 \rfloor - (s - t))$ ticks.

Proof. Lemma 6 establishes the invariance of bumpiness. If after $\ell' < \ell$ rounds, a smooth state σ' is obtained, appeal to Lemma 13 for that part of the computation following round ℓ'. Therefore consider only the case where each state of B during the first ℓ rounds is bumpy, but not smooth, and there is at least one safe region S that has border processes. The remainder of the proof has three claims: (*0*) define $bdist_p$ for $p \in S$ to be the smallest distance from p to a border process in S; then after one round, $bdist_p$ increases (this can be shown by contradiction); (*1*) let P be the set of all border processes of S, that is, $(\forall r : r \in P : \mathsf{clock}_p = 0)$ at state σ; if each process in P increments its clock at least $\lfloor \ell/2 \rfloor$ times within ℓ rounds, then for each $p \in S$, the value of clock_p increments at least $\lfloor \ell/2 \rfloor - s_p$ times within ℓ rounds, where s_p denotes the value of clock_p at state σ (the claim follows essentially from the same argument given for the proof of Lemma 13); (*2*) each border process of S increments its clock at least $\lfloor \ell/2 \rfloor$ times within ℓ rounds (this can be shown by induction). □

Lemma 15. Let B be a computation of bumpy states. If at any state $\sigma \in B$, process p satisfies $\mathsf{clock}_p < \mathcal{T}$, and p observes ℓ ticks following σ, each process q observes at least $\ell - 2dist_{pq}$ ticks.

Proof. The lemma can be shown by induction on distance, and for a given distance nested induction on ℓ is used. □

Lemma 16. Let B be a computation and let p be a process in a safe region at the initial state of B. If at any state $\sigma \in B$, process p satisfies $\mathsf{clock}_p < \mathcal{T}$, and p observes ℓ ticks following σ, each process q observes at least $\ell - 2dist_{pq}$ ticks.

Proof. The proof is the same as that for Lemma 15, joining to the induction the fact that safe regions grow at each round, or merge with other safe regions as observed in Lemmas 3 and 4. □

5.3 Output Stabilization

Lemma 17. Every computation originating from a k-faulty state contains a bumpy state within $3 + \min(\mathcal{D}, k)$ rounds.

Proof. If $k \geq \mathcal{D}-1$, then Lemma 5 implies the result; therefore suppose $k < \mathcal{D}-1$. Recall that in a k-faulty state, each non-faulty process p has $\mathsf{clock}_p = \mathcal{T}$ initially, so the first change to clock_p is a reset to zero. After two rounds, the computation satisfies the neighbor relations of Lemma 1. After the third round, any process observing a difference in clock values greater than 1 resets its clock. Once a clock is assigned zero, it remains in some safe region throughout the computation.

It is easy to show that if no faulty process resets its clock, a final state is obtained within $2 + k$ rounds. However if any process resets its clock in the computation, the claim is that some process p resets its clock by round three, and p is either initially faulty or a neighbor of a faulty process. These arguments apply independently to each faulty region in the initial state, so each faulty region either converges to a final value (i.e., the clock reaches $\mathcal{T}$ for each process), or some process neighboring to or within the faulty region resets its clock by the end of round three. Once a zero-valued clock exists, processes in the faulty region join a common safe region at the rate of at least one diameter unit per round, and therefore after k additional rounds, all initially faulty processes are in a safe region. □

Lemma 18. The repair timer construction satisfies Requirement 1 within $3 + \min(\mathcal{D}, k)$ rounds for any computation starting from a k-faulty initial state.

Proof. Lemmas 17 and 15 imply the result. □

Lemma 19. Let B be a computation starting from a k-faulty initial state, $k > 0$. Within $3+\min(\mathcal{D}, k)$ rounds of B there is a state σ such that Requirement 1 holds for the remainder of B, and following σ each process observes at least $\mathcal{C}_{\mathcal{D}}^{-1}(g \cdot \mathcal{D})$ ticks within $O(\mathcal{D})$ rounds, thus satisfying Requirement 3.

Proof. Lemma 17 identifies σ, and the proof of Lemma 17 applies to the case $k = n$, which is the worst-case faulty initial state. So σ is obtained within $3 + \mathcal{D}$ rounds, even for $k < n$. Also, the proof of Lemma 17 is based on the assumption that some process resets its clock in the first three rounds, thereafter remains in a safe region, and that this region grows at the rate of one process per round. When the safe region grows to cover the network, a smooth state σ is obtained with $\mathcal{D}$ as the maximum clock value. By Lemma 15 subsequent computation satisfies Requirement 1. To guarantee that each process observes at least $\mathcal{C}_{\mathcal{D}}^{-1}(g \cdot \mathcal{D})$ ticks, it is enough to establish that $\mathcal{T} - \mathcal{D} \geq \mathcal{C}_{\mathcal{D}}^{-1}(g \cdot \mathcal{D})$, since all clock variables increment to the target value of $\mathcal{T}$ within $O(\mathcal{D})$ rounds by Lemma 8. This inequality is directly verified by substitution of the definitions of $\mathcal{T}$ and $\mathcal{C}$. □

Lemma 20. Let B be a computation starting from a k-faulty initial state σ, and let X be a faulty region at state σ. If any process $p \in X$ resets clock_p in the first three rounds of B, then within the first $3 + \min(\mathcal{D}, k)$ rounds of B there is a safe state where each process $q \in X$ satisfies $\mathsf{clock}_q \leq 2k$.

Proof. The proof of Lemma 17 has an argument showing that if some process in a faulty region resets its clock, then within $3 + k$ rounds there exists a state where each initially faulty process is in a safe region. The claim needed here is that once process p resets its clock, it does not increase clock_p beyond the value k until p is contained in a safe region of diameter at least k — such a safe region of diameter at least k contains X and by the definition of a safe region, each clock in X is at most $2k$. □

Lemma 21. Let B be a computation starting from a k-faulty initial state σ and let X be a faulty region at state σ, and let Y be the smallest region containing X such that each non-faulty process in Y is neighbor to a process in X. If any process $p \in X$ resets clock_p in the first three rounds of B, then within $3+\min(\mathcal{D}, k)$ rounds there exists a state $\delta \in B$ with the following properties. For the suffix computation C beginning at δ, for each process $q \in Y$, Requirement 1 holds for q throughout C; also, for each faulty process q in Y, $\mathsf{clock}_q \leq 2k$ at state δ, and for each non-faulty process q in Y, either $\mathsf{clock}_q \leq 2k$ or $\mathsf{clock}_q = \mathcal{T}$ at state δ. Finally, within $1+2(h+4)k$ rounds of C, each process $q \in Y$ observes at least $\mathcal{C}_k^{-1}(h \cdot k)$ ticks, provided $k \leq \mathcal{D}$ and $g > 2h + 7$.

Proof. Lemma 20 provides the starting point for the proof by establishing that within $3+\min(\mathcal{D}, k)$ rounds there is a state δ such that all processes in X belong to a common safe region with clock at most $2k$. Lemma 17 states that by the end of round $3 + \min(\mathcal{D}, k)$ the state is bumpy. This implies that either δ is bumpy or some state γ occurring later in the round is bumpy. Although some clock in Y may exceed $2k$ when state γ is reached, arguments based on γ will obtain a conservative estimate on the number of additional rounds needed for the lemma's conclusion.

Lemma 14 states that each process in X observes at least $\min(\mathcal{T} - s, \lfloor \ell/2 \rfloor - (s - t))$ ticks within ℓ rounds, where s and t refer to current and minimal clock values at state γ. To obtain a rough bound, suppose $t = 0$ and $s = 2k$. Then to require a process of X to observe $\mathcal{C}_k^{-1}(h \cdot k) = h \cdot k + 2k$ ticks,

$$\min(\mathcal{T} - 2k, \lfloor \ell/2 \rfloor - 2k) \geq h \cdot k + 2k \;\Rightarrow\; \ell > 2(h + 4)k$$

The remaining question to complete the proof is to verify that $\mathcal{T} - 2k$, the number of ticks that a faulty process observes following state γ, is sufficient for the required $2(h + 4)k$ bound. Observe that $\mathcal{T} - 2k > 2(h + 4)k$ reduces to $g > 2h + 7$ if $k \leq \mathcal{D}$ is assumed, which establishes the result. ◻

5.4 Integrating Repair Timer Methods

Results proven to this point depend on some clock variables being reset early in a computation; for instance Theorem 1 and Lemma 20 use the condition that a faulty clock resets within the first three rounds of a computation (also sufficient would be a faulty clock initially zero). Two questions arise when integrating the repair timer into the fault model described in Section 2. First, it should be that some clock does reset early enough in a computation to support the conditions of Theorem 1. Secondly, it should not be that clock variables continue to reset indefinitely.

The model of fault behavior described in Section 2 specifies that in the first round of any computation, for any faulty region X, some process p in X detects a fault condition and calls start, which ensures $\mathsf{clock}_p < \mathcal{T} - \mathcal{D}$. If calling start makes X smooth, then some non-faulty neighbor will reset in the next round, and if X is non-smooth, some process of X will reset in the next round.

After some clock resets within a faulty region X, a safe region is established, which grows in diameter at each round — provided no other clock resets interfere. However any additional start calls fall into two possible cases: either the start call occurs within a safe region established (and enlarged) from a clock reset in the first two rounds, or the start call occurs at some process of X that had not previously detected a fault condition. If the latter case resets a clock, it also contributes to progress toward the condition that all clock values are at most $2k$ within the first $3 + \min(\mathcal{D}, k)$ rounds (see Lemma 20). Within $O(k)$ rounds, all clock values are at most $2k$ within X and X is a safe region. Any start calls after this point do not change clock values, so long as start calls cease before clock variables increment to $\mathcal{T} - \mathcal{D}$. Repairs complete within $h \cdot k$ rounds, so if at least $h \cdot k$ rounds elapse before clock values reach $\mathcal{T} - \mathcal{D}$, then additional start calls do not occur. The condition to verify is therefore: $(\mathcal{T} - \mathcal{D}) - 2k \geq h \cdot k$, which is $(g + 2)\mathcal{D} \geq (h + 2)k$. Theorem 1 specifies $k \leq \mathcal{D}$, so all that is needed is $g \geq h$, which is conveniently also implied by Theorem 1's conditions.

References

1. A Arora, S Dolev, and MG Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.
2. B Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32:804–823, 1985.
3. JM Couvreur, N Francez, and MG Gouda. Asynchronous unison. In *ICDCS92 Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 486–493, 1992.
4. S Dolev. Optimal time self-stabilization in dynamic systems. In *WDAG93 Distributed Algorithms 7th International Workshop Proceedings, Springer-Verlag LNCS:725*, pages 160–173, 1993.
5. S Even and S Rajsbaum. Unison, canon and sluggish clocks in networks controlled by a synchronizer. *Mathematical Systems Theory*, 28:421–435, 1995.
6. S Ghosh, A Gupta, T Herman, and SV Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, 1996.
7. S Ghosh, A Gupta, and SV Pemmaraju. A fault-containing self-stabilizing algorithm for spanning trees. *Journal of Computing and Information*, 2:322–338, 1996.
8. MG Gouda and T Herman. Stabilizing unison. *Information Processing Letters*, 35:171–175, 1990.
9. T Herman. Superstabilizing mutual exclusion. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95)*, pages 31–40, 1995.
10. T Herman. Distributed repair timers. Technical Report TR 98-05, University of Iowa Department of Computer Science, 1998.
11. T Herman and S Ghosh. Stabilizing phase-clocks. *Information Processing Letters*, 54:259–265, 1995.
12. S Kutten and B Patt-Shamir. Time-adaptive self stabilization. In *PODC97 Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 149–158, 1997.
13. C Lin and J Simon. Possibility and impossibility results for self-stabilizing phase clocks on synchronous rings. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 10.1–10.15, 1995.
14. J Misra. Phase synchronization. *Information Processing Letters*, 38:101–105, 1991.