Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Self-Stabilizing Wait-Free Clock Synchronization

M. Papatriantafilou, P. Tsigas

Computer Science/Department of Algorithmics and Architecture

CS-R9421 1994

Self-Stabilizing Wait-Free Clock Synchronization

Marina Papatriantafilou
Email: ptrianta@cwi.nl

Philippas Tsigas
Email:tsigas@cwi.nl

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
& Computer Technology Institute, Patras, Greece
& Computer Science Dept., University of Patras, 26500 Patras, Greece

Abstract

Clock synchronization algorithms which can tolarate any number of processors that can fail by ceasing operation for an unbounded number of steps and then resuming operation (with or) without knowing that they were faulty are called Wait-Free. Furthermore, if they are also able to work correctly even when the starting state of the system is arbitrary, they are called Wait-Free, Self-Stabilizing. This work deals with the problem of Wait-Free, Self-Stabilizing Clock Synchronization of n processes in an "in-phase" multiprocessor system and presents a solution with synchronization time $O(n^2)$. The best previous solution has $O(n^3)$ synchronization time. The idea of the algorithm is based on a simple analysis of the difficulties of the problem which helped us to see how to "reparametrize" the $O(n^3)$ previously mentioned algorithm in order to get the $O(n^2)$ synchronization time solution. Both the protocol presented here and its analysis are very simple.

AMS Subject Classification (1991): 68M10, 68Q22, 68Q25
CR Subject Classification (1991): D.4.1, D.4.5, D.4.7
Keywords & Phrases: Concurrency, Digital Clocks, Distributed Computing, Fault tolerance, PRAM computation model, Self-Stabilization, Synchronization, Synchronous Systems, Wait-Free Synchronization
Note: This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 7141 (project ALCOM II). The first author was also partially supported by a NUFFIC Fellowship. The second author was also partially supported by NWO through NFI Project ALADDIN under contract number NF 62-376.

1. Introduction

Synchronization among the processes of a multi-processor system is commonly obtained using clocks. In general a clock is implemented in a multi-processor system in one of the following ways: i) using a single clock that is connected to all the processors in the system, ii) using individual clocks for every processor that are connected to a pulse generator which generates clock pulses stimulating the individual clock, iii) using individual clocks and pulse generators for each processors. It is easy to see that the less centralized the clock implementation is the more resilient to faults it is.

In the past clock synchronization solutions that can tolerate faults have been proposed for the case of arbitrary, or Byzantine, faults [19, 18, 20, 8, 21, 23]. In those model characteristics they proved that no algorithm can work unless more than one third of the processors are nonfaulty [8]. In the case of authenticated Byzantine faults the things are not so bad; there exist algorithms that can tolerate any number of faulty processors [12]. The negative results in that model are that: i) the faulty processors can influence the clocks of the non-faulty ones by speeding them up, ii) reaccession of repaired processors is not possible unless more than half of the processors are non-faulty [12]. *Self-stabilizing* algorithms for the clock synchronization problem have also been proposed [11, 6, 1]. An algorithm is called *self-stabilizing* if it can tolerate *transient faults* in the sense that, after a transient fault leaves the system in an arbitrary state, if no further fault occurs for a sufficiently long period of time then the system converges into a consistent global state and can solve the task. A *transient fault* is a fault that causes the state of a process (its local state, programm counter and its shared variables) to change arbitrarily. More about self-stabilization can be found in e.g. [7, 2, 9, 5, 4, 22].

So, if we want to sum it all up, the "ideal" clock synchronization algorithm that is highly resilient to failures must have the following characteristics: (i) it must tolerate any number of processors' *napping faults* like the authenticated Byzantine model *but* guarantees that the nonfaulty processors' clocks remain unaffected by the failures, (ii) faulty processors are able to rejoin the system and become synchronized in a number of k steps that is indepentent of the number of the working processors, and (iii) it works correctly regardless of the system state in which it is started.

Recently Dolev and Welch in [10] presented this highly resilient view of clock synchronization as *Wait-Free, Self-Stabilizing Clock Synchronization.* The assignement of this name to the problem is due to the facts that the first two conditions mentioned in the previous paragraph capture the spirit of the *wait-freedom* (cf., e.g., [16, 3, 13]) in the presence of *napping faults* and the third condition captures the spirit of *self-stabilization.* In that paper they present two *Wait-Free, Clock Synchronization* algorithms for n processors which assume a global clock pulse ("in-phase" systems) and nonglobal read/modify/write atomicity. Those solutions guarantee synchronization within $O(n^3)$ and $O(n^2)$ steps; the first solution is also a *Self-Stabilizing* one, while the second depends on the initialization.

In this paper we examine the same problem. By pointing out a simple approach in analyzing the difficulties of the problem, we show how to "reparametrize" the $O(n^3)$ algorithm of [10], thus getting a solution to the Clock Synchronization problem which is both *Wait-Free* and *Self-Stabilizing*, and has synchronization time $O(n^2)$. Moreover, its analysis and proof of correctness are simple and intuitive.

2. The Model

The system consists of n identical processors. A processor p_i is a (possibly infinite) machine. The processors communicate via a set of single-writer, multi-reader atomic registers. Each processor owns a subset of these registers. The owner of a register can write the register while all the other processors can read it. A *step* by a process p_i consists of the following actions: (i) read by p_i of the shared registers owned by some particular processor p_j ($i \neq j$), (ii) transition of p_i's local state (program counter, local variables), and (iii) update of its own shared registers.

We consider "in-phase" systems, in which all processors share a common clock pulse. Each pulse is a (possibly empty) set of processor names; the set of processors that *make a step* in the pulse. Each processor can make at most one step in one pulse. If a processor does not make a step in some pulse it will be said to *take a pause*.

A *configuration* is a tuple of processors' states and of values of the shared variables. A system *execution* is a sequence $c_0\pi_1c_1\pi_2\ldots$ of alternating pulses (denoted by π_x) and configurations (denoted by c_x). Pulses indexed with consecutive numbers will be called *consecutive*. Each configuration c_i in a system execution is derived from its directly preceding configuration c_{i-1} by the state transitions and the shared variables' updates of the processors that make a step in the pulse π_i in between these configurations; the shared registers' reads by all the processors that make a step in π_i return the respective values of c_{i-1}, while the shared registers' updates take place in unison to derive c_i. An execution is *initialized* if its first configuration is explicitly specified by the protocol. We will refer to a sub-sequence (starting and ending with a configuration) of the sequence which describes a system execution by the term *sub-execution* of that execution. The *length* of a sub-execution is the number of pulses in it. In a sub-execution s' (with length greater or equal to l) of a system execution s, a processor p_i will be said to have made l *continuous steps* if it makes steps for l *consecutive* pulses of s'.

This system can be viewed as modeling either a PRAM (cf. [17, 15]) with faults or a multiprocessor synchronous system (cf. [14]) in which scheduling of the processes in different processors is done independently. Pause intervals can be interpreted as periods during which some process is not scheduled in a processor, or as faults in the connections of the pausing proccessor or as transient faults, or even as processor crashes.

In a solution to the clock synchronization problem, each processor owns a shared variable which holds the value of its clock. The requirement from a wait-free clock synchronization algorithm is that there should be a positive integer k such that for any execution s of the protocol:

- ADJUSTMENT: For any $l > k$ and for any processor p_i that makes l continuous steps during a sequence of consecutive pulses $\pi_{j+1}, \ldots, \pi_{j+l}$, p_i's clock in c_{j+l} equals its clock in c_{j+l-1} incremented by one.
- AGREEMENT: For any $l \geq k$ and for any two processors p_i and p_j that have both made l continuous steps during any sequence of pulses $\pi_{j+1}, \ldots, \pi_{j+l}$, p_i's and p_j's clocks in c_{j+l} are equal.
- If self-stabilization should also be guaranteed by the solution, then the above two requirements should be met even in non-initialized executions.

3. The Protocol

3.1 Informal Description

First we will try to give an insight into the characteristics of the problem by applying an easy strategy: each processor which has possibly taken a pause tries to catch up with the

```
var (CLOCK_1, CNT_1), ..., (CLOCK_n, CNT_n): (int, int) ;
    /* Shared variables declaration*/

Synch(i) /* version for process i */
var j, clock_j, cnt_j, diff, my_clock, my_cnt, susp: int ;
    prev: array [1..n] of int ;
begin
    repeat
        for j = 1 to n (j ≠ i) do
            read (CLOCK_j, CNT_j) into (clock_j, cnt_j) ;
            my_cnt := CNT_i + 1 ;
            diff := cnt_j − prev[j] ;   prev[j] := cnt_j ;
            if susp ≠ 0 then susp := susp − 1 ;
            if diff > n − 1 then susp := 2n(n − 1) ;
            if susp = 0 then my_clock := max(clock_j, CLOCK_i) + 1 ;
                        else my_clock := CLOCK_i ;
            write (my_clock, my_cnt) to (CLOCK_i, CNT_i) ;
        end_for
    forever
end
```

Figure 1: The Protocol

maximal clock in the system, by scanning in cyclic order the other processors' clocks and by simply updating its own clock to the maximum clock value it sees in each step. In schedules in which for a period of time only one process (not necessarily the same during the period) holds the maximal clock value in the system, we can think of the maximal value as a "ball" which is "passed" from one process to the other, under a proper interleaving of their working steps and their pauses. Now, suppose that there exists a process p_i which tries to find the maximal clock value and which does not take any pauses, which implies that within a certain number of steps it should achieve its goal. However, there might be a set S of other processes (more than two) which are scheduled (take pauses or make steps) so that each one p_x of them does not hold the maximal clock value at the pulses when its clock is read by p_i but reads that value from another process in S immediately after its own value has been read by p_i; then it keeps and increments that value for a number of pulses that are not enough for p_i to complete a cycle and read p_x's clock again; in the meantime another process p_y can do the same as p_x did. This "game" can be played by all the processes in S scheduled in a way that they cyclically take turns in misleading p_i and preventing it from catching up with the maximal clock in the system. The duration of such a game can be infinite, but the game is also "stop-able" at any time, which implies that at any time it will be possible for p_i to violate the *adjustment* requirement.

The protocol presented here —which is a reparametrized modification of the protocol presented in [10]— protects the correctly working processors in the following way: each process repetetively scans the clock values of the other processes in cyclic order, trying to keep up with the most advanced of them. When a processor p_i has taken some pause and its clock needs adjustment, it is guaranteed that after it has made a certain number of continuous steps its own clock will be as far as $n-1$ or less from the maximal clock value of the system at that time. After that, what p_i needs from the schedule in order to find the maximal clock value, is either (i) some process which holds t he maximal clock value to continuously keep making steps for as long as a scan takes ($n-1$ steps) or (ii) a slow-down of the incrementing of the maximal clock value by $n-1$ steps. The former will happen if that process correctly makes steps. Towards the latter, each processor which misleads p_i (necessarily by taking a pause) is suspended (does not increment its clock) for a period of time until p_i has safely (by the pigeon-hole principle) found the maximal clock value. Suspension is implemented with the use of a local variable *susp* for each process. Moreover, each process can detect whether it paused or not by checking its relative speed with respect to the other processors. This mechanism is implemented with the use of the shared variable CNT_i and the local array *prev* by each process p_i.

At this point it should be mentioned that, as proven in [10], there can be no wait-free, self-stabilizing clock synchronization algorithm with only *blind* write operations (i.e. updates of its shared variables by p_i without prior reading them). In the protocol described here, it can be easily seen that p_i never performs a blind write.

The formal description of the protocol is given in Figure 3.1.

3.2 Proof of Correctness

We will first show that the protocol described meets the requirement of a solution to the wait-free clock synchronization problem: for any processor p_i ($1 \leq i \leq n$) which is working correctly (performs continuously steps without taking pauses in between) for at least $k = (4n+1)(n-1)$ pulses, as long as it continues working correctly, its clock will not need adjustment and will agree with the clock of any other processor which has been working correctly for at least k pulses. Towards that we will first prove that p_i after at most k continuous steps will be guaranteed to hold the maximal clock value in the respective system's configuration. Some auxiliary definitions will help the presentation of our arguments:

NOTATION 1 *If c denotes a system* configuration *then $CLOCK_i(c)$ denotes the value of the respective shared register in c. Besides, $MAX\_CLOCK(c)$ denotes $max\{CLOCK_i(c) : 1 \leq i \leq n\}$.*

DEFINITIONS 1 *:*

- A process p_i ($1 \leq i \leq n$) is *suspended* in some configuration in a system execution if its local variable $susp \neq 0$ in that configuration.
- An *adjustment phase* for a process p_i in a system execution s is a subexecution $s' = c_j \pi_{j+1} c_{j+1} \ldots \pi_{j+l} c_{j+l}$, such that:
  1. p_i makes a step in all the pulses in s' and in pulse π_{j+l+1} of s it takes a pause.

2. the local variable *susp* of p_i equals 0 in all the configurations in s'.
3. c_j is either the first configuration of s or there exists π_j in which p_i either takes pause or makes a step in which it changes the value of its local variable *susp* from 1 to 0.

- A process p_i performs a *forwarding step* in a particular pulse π_j in some system execution if $CLOCK_i(c_{j-1}) < CLOCK_i(c_j)$ and $CLOCK_i(c_j) = MAX\_CLOCK(c_j)$, where c_{j-1} and c_j are the system configurations directly preceding and immediatelly following that pulse. A pulse in an execution is *forwarding* if there exists a process p_i which makes a forwarding step at that pulse; otherwise we will call the pulse *non-forwarding*.
- A *round* of a process p_i is a sequence of $n-1$ successive steps by p_i.(In a round a processor reads the shared information of all the other processors in the system.)

It can be easily seen that if c_{j-1} and c_j are the system configurations directly preceding and immediatelly following a pulse π_j, then either $MAX\_CLOCK(c_j) = MAX\_CLOCK(c_{j-1})+1$ or $MAX\_CLOCK(c_j) = MAX\_CLOCK(c_{j-1})$ depending on whether the pulse is forwarding or non-forwarding, respectively.

Assume that a process p_i makes at least $k = (4n+1)(n-1)$ continuous steps in continuous pulses in a system execution. In the following lemmas we prove that at most by the last of these steps it will hold the maximal clock value in the system.

Lemma 1 *In the configuration c after the last pulse of a sequence of $(2n+1)(n-1)$ continuous steps by a process p_i in a system execution its local variable susp will equal* 0*.*

Proof. In the first round of p_i in the sequence defined, p_i will load its array *prev* with the value of the CNT_x shared variable of each other process p_x. Even if in that round p_i becomes suspended (its local variable *susp* is assigned the valued $2n(n-1)$) —due to the fact that prior to these steps that array could contain arbitrary values—, in the next rounds the computation of its local variable $diff$ $(\leq n-1)$ will result in decrementing the value of *susp*, which implies that by the last step of the sequence, *susp* will equal 0. □

The above lemma implies that at most after its first $(2n+1)(n-1)$ continuous steps p_i will enter an adjustment phase, which, due to our assumption for p_i, is going to last at least $2n(n-1)$ pulses. During the adjustment phase and if there are no transient faults in the system, its local variable *susp* will never become non-zero and the value of $CLOCK_i$ will be incremented by at least 1 at each pulse.

Lemma 2 *In the configuration c after the first round of p_i in an adjustment phase in a system execution it will hold that $MAX\_CLOCK(c) - CLOCK_i(c) \leq n-1$. Moreover, for any sequence of l ($l \leq 2n(n-1)$) continuous steps of p_i in its adjustment phase, if c_j and c_{j+l} are the configurations directly preceding the first and immediately following the last pulse of the sequence and if $d_j = MAX\_CLOCK(c_j) - CLOCK_i(c_j)$ and $d_{j+l} = MAX\_CLOCK(c_{j+l}) - CLOCK_i(c_{j+l})$ it will hold that $d_j \geq d_{j+l} + l_{nf}$, where l_{nf} is the number of non-forwarding pulses during the specified sequence of l steps.*

Proof. For the first part of the lemma let c^- denote the configuration directly preceding the first step of p_i in the round specified. Then it holds that $MAX\_CLOCK(c) -$

$MAX\_CLOCK(c^-) \leq n-1$ because at each step the maximal clock of the system can be increased by at most one. But $MAX\_CLOCK(c^-)$ is the value of $CLOCK_x$ in c^- for some process p_x in the system, which p_i is going to read in one of the steps of the round. Since the values of the $CLOCK$ variables are never decremented it follows that: $CLOCK_i(c) \geq MAX\_CLOCK(c^-)$. This inequality implies that: $MAX\_CLOCK(c) - CLOCK_i(c) \leq MAX\_CLOCK(c) - MAX\_CLOCK(c^-)$, which, combined with our first inequality, implies that $MAX\_CLOCK(c) - CLOCK_i(c) \leq n-1$.

The inequality of the second part of the lemma can be derived by direct combination of the following two statements: (i) $CLOCK_i(c_{j+l}) \geq CLOCK_i(c_j) + l$ because p_i is not suspended and, thus, it increments its clock by at least one in each step. (ii) $MAX\_CLOCK(c_{j+l}) = MAX\_CLOCK(c_j) + l - l_{nf}$, because the system's maximal clock is incremented by one in each pulse, unless the pulse is non-forwarding. □

The previous lemma states that once p_i enters the adjustment phase, after the first round it is guaranteed to have a clock value which differs by at most $n-1$ from the maximal clock value of that configuration and that this difference can only decrease in the following steps of p_i. Hence, we have the following:

Lemma 3 *Assume that an adjustment phase of a processor p_i with length at least $2n(n-1)$ pulses in a system execution and consider the subexecution which starts with the system configuration after the first round of p_i in the phase and ends with the configuration after the $2n(n-1)$-th step of p_i in the phase. If in this subexecution there are $n-1$ or more non-forwarding pulses, then it will hold that $CLOCK_i(c) = MAX\_CLOCK(c)$, where c is the last configuration of the subexecution.*

Proof. It follows from Lemma 2 and from a fact that is directly derived from the rules of the protocol: if p_i at some step reads the maximal clock value of that configuration then,as long as p_i continues working correctly it will still hold the maximal clock value in the system and that it will increment its clock by one at each pulse. □

Lemma 4 *If the length of an adjustment phase of p_i is at least $2n(n-1)$ pulses in a syetem execution then at the configuration c after the $2n(n-1)$-th step of the phase it will be the case that $CLOCK_i(c) = MAX\_CLOCK(c)$.*

Proof. We make the assumption, towards coming to a contradiction, that $CLOCK_i(c) < MAX\_CLOCK(c)$. Let A denote the subexecution specified by the first $2n(n-1)$ steps of p_i in this adjustment phase. Also, consider any process p_x $(x \neq i)$ which makes steps during A. We make two crucial remarks:
(i) Under our assumption, p_x cannot perform $n-1$ continuous forwarding steps during A. Otherwise, we already have a contradiction: Since $CLOCK_x$ is read by p_i every $n-1$ steps and because p_i's steps in the specified interval are continuous by definition, p_i would have adjusted its own clock to $CLOCK_x$ and, hence to the maximal clock of the system during one of these $n-1$ steps of p_x.
(ii) Once p_x performs its first $n-1$ steps (not necessarily continuous) in A, it will load its local variable $prev[i]$ with a correct value of CNT_i written by p_i during A; thus, p_x will have a consistent reference time-point for detecting its pauses thereafter. After that point, due to

our assumption, p_x cannot make more than $n-1$ forwarding steps in A: if it does, we know from (i) that these steps will not be continuous. But then, by at most the $(n-1)$-th such step it will detect its pause, and, as a result it will become suspended. Since the length of a subexecution in which a processor is continuously suspended is at least equal to the duration of A ($2n(n-1)$ pulses), p_x will not increment its clock again during A.

What (ii) essentially implies is that the number of forwarding steps of each process p_x ($x \neq i$) during A is at most $2(n-1)$, which means that the total number of forwarding pulses in A is at most $2(n-1)^2$. The latter in turn implies that the number of non-forwarding pulses during A is at least $2(n-1)$ and, in particular after p_i's first round in A it is at least $n-1$. But then, by Lemma 3 p_i should hold the maximal clock value at c, which contradicts our assumption. □

THEOREM 1 *The construction correctly implements a self-stabilizing wait-free clock synchronization solution with $k = (4n+1)(n-1)$.*

PROOF. After a process p_i has worked correctly for at least $k = (4n+1)(n-1)$ steps, it is guaranteed by Lemma 4 that it will hold the maximal clock value in the system. After that, it can be directly derived from the rules of the protocol, that as long as it continues working correctly it will still hold the maximal clock value in the system and that it will increment its clock by one at each pulse. The same will hold with any other process that has been working continuously and correctly for at least k pulses; this implies that its clock value will agree with the clock value of p_i.

The self-stabilizing property of the protocol is due to the facts that in the analysis (i) no initialization conditions are needed and (ii) it is shown that after transient faults have ceized, each process which performs k continuous steps will converge to legal behaviours, as defined by our requirements of a solution to this problem. □

CONCLUSIONS

In this work we show a wait-free and self-stabilizing protocol that achieves clock synchronization among n processors in at most $(4n+1)(n-1)$ steps, and which improves the previously known solution which had synchronization time $O(n^3)$ steps. The best known non-self-stabilizing solution to the same problem also has synchronization time $O(n^2)$. However, the question whether the problem can be solved with a linear time algorithm it is still open. Another point that deserves consideration is whether the requirement for self-stabilization imposes an overhead in the complexity of the problem.

ACKNOWLEDGMENT

We are thankful to Moti Yung for his help in the first steps of this work.

REFERENCES

1. A. ARORA, S. DOLEV, AND M. GOUDA. Maintaining Digital Clocks in Step. *Parallel Processing Letters 1*, 1, 1991, pp. 11-18.
2. Y. AFEK, S. KUTTEN AND M. YUNG. Memory–Eficient Self Stabilization on General Networks. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, volume 486 of *Lecture Notes in Computer Science*, Springer-Verlag 1990, pp. 15–28.

3. H. Attiya, N. A. Lynch and N. Shavit. Are Wait-Free Algorithms Fast? In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science,* 1990, pp. 55–64.

4. B. Awerbuch, B. Patt-Shamir and G. Varghese. Local Checking and Correction. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, 1991, pp. 268–277.

5. B. Awerbuch and G. Varghese. Distributed Program Checking: A Paradigm for Building Self-Stabilizing Distributed Protocols, In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, 1991, pp. 258–267.

6. J.-M. Courver, N. Francez and M. Gouda. Asynchronous Unison. In *Proceedings of the 12th IEEE Conference on Distributed Computing Systems,* 1992, pp. 486–493.

7. E.W. Dijkstra. Self Stabilizing Systems in Spite of Distributed Control. *Communication of the ACM 17*, 1974, pp. 643–644.

8. D. Dolev, J.Y. Halpern and H.R. Strong. On the Possibility and Impossibility of Achieving Clock Synchronization. *Journal of Computer Systems Science 32*, 2, 1986, pp. 230–250.

9. S. Dolev, A. Israeli and S. Moran. Self Stabilization on Dynamic Systems Assuming Only Read/Write Atomicity. *Distributed Computing 7*, 1, 1993, pp. 3–16.

10. S. Dolev and J.L. Welch. Wait-Free Clock Synchronization. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing,* 1993. pp. 97–108.

11. M.G. Gouda and T. Herman. Stabilizing Unison. *Information Processing Letters 35*, 1990, pp. 171–175.

12. J. Halpern, B. Simons, R. Strong And D. Dolev. Fault-Tolerant Clock Synchronization. In *Proceedings Of the 3rd ACM Symposium on Principles of Distributed Computing*, 1984, pp. 89–102.

13. Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (Jan. 1991), pp. 124–149.

14. K. Hwang. *Advanced Computer Architectures, Parallelism, Scalability, Programmability.* McGraw-Hill, Inc. 1993.

15. R. Karp and V. Ramachandran. Parallel Algorithms for Shared Memory Machines. In J.van Leeuwen, ed., *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity* Elsevier, Amsterdam 1990. Also in: Technical Report UCB/CSD 88/408, Computer Science Division, University of California, March 1988.

16. L. Lamport. On Interprocess Communication. *Distributed Computing 1*, 1, 1986, pp. 86–101.

17. F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, Inc., 1992.

18. L. Lamport and P.M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *Journal of the ACM 32,* 1, 1985, pp. 1–36.

19. K. Marzullo. *Loosely-Coupled Distributed Services: A Distributed Time Service,* Ph.D. Thesis, Stanford University, 1983.

20. S. MAHANEY AND F. SCHNEIDER. Inexact Agreement: Accuracy, Precision and Graceful Degradation. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing,* 1985, pp. 237–249.

21. T.K. SRIKANTH AND S. TOUEG. Optimal Clock Syncronization. *Journal of the ACM 34,* 3, 1987, pp. 626–645.

22. G. VARGHESE. *Self-Stabilization by Local Checking and Correction.* Ph.D. Thesis, MIT Laboratory for Computer Science, 1992.

23. J.L. WELCH AND N. LYNCH. A New Fault-Tolerant Algorithm for Clock Synchronization. *Information and Computation 77,* 1, 1988, pp. 1–36.