On Self-Stabilizing Wait-Free Clock Synchronization*

MARINA PAPATRIANTAFILOU (Email: ptrianta@mpi-sb.mpg.de)
Max Planck Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany
& CE and Informatics Dept., Patras University, Greece

PHILIPPAS TSIGAS (Email: tsigas@mpi-sb.mpg.de)
Max Planck Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

March 26, 1996

Abstract

Protocols which can tolerate any number of processors failing by ceasing operation for an unbounded number of steps and resuming operation (with or) without knowing that they were faulty are called *wait-free*; if they also work correctly even when the starting state of the system is arbitrary, they are called *wait-free, self-stabilizing*. This work is on the problem of *wait-free, self-stabilizing clock synchronization* of n processors in an "in-phase" multiprocessor system and presents a protocol that achieves quadratic synchronization time, by "re-parameterizing" and improving the best previously known solution, which had cubic synchronization time. Both the protocol and its analysis are intuitive and easy to understand.

1 Introduction

SYNCHRONIZATION among the processors of a multi-processor system is commonly obtained using logical clocks. Since by today's technology multiprocessor systems have large numbers of processors and since the probability of failure increases with the number of processors in the system, it is important both to study which multiprocessor models can support protocols that tolerate faults, as well as to design such fault-tolerant protocols for them.

In the past clock synchronization solutions that can tolerate faults have been proposed for the case of arbitrary, or Byzantine faults [4, 13, 14, 15, 16, 18]. In those system models it has been proven that no algorithm can work unless more than one third of the processors are non-faulty [4]. In the case of authenticated Byzantine faults the situation is not so bad; there exist algorithms that can tolerate any number of faulty processors [7]. The negative results in that model are: i) the faulty processors can influence the clocks of the non-faulty ones by speeding them up, ii) re-accession of repaired processors is not possible unless more than half of the processors are non-faulty [7]. *Self-stabilizing* algorithms for the clock synchronization problem have also been proposed [1, 2, 6]. 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

*A preliminary version of this work appeared in the Proceedings of the 4th Scandinavian Workshop on Algorithm Theory (SWAT'94), Lecture Notes in Computer Science Vol. 824, Springer-Verlag, 1994.

converges into a consistent global state and can solve the task. For an introduction and a survey on self-stabilization, see [3, 17].

So, if we want to sum it all up, the "ideal" clock synchronization algorithm that is highly resilient to failures must have the following features: (i) it must not only tolerate any number of processors' *napping faults* like the authenticated Byzantine model but also guarantee that the non-faulty processors' clocks remain unaffected by the failures, (ii) it must allow processors which have been faulty to rejoin the system when they resume normal operation and become synchronized in a number of steps k (*synchronization time*) independent of the number of the working processors, and (iii) it must work correctly regardless of the system state in which it is started.

Recently Dolev and Welch in [5] presented this highly resilient view of the problem as *wait-free, self-stabilizing clock synchronization*; the first two conditions mentioned above capture the spirit of the *wait-freedom* (cf. e.g. [8, 11]) which implies maximum resiliency to processor *halt/napping failures* and the third condition captures the spirit of *self-stabilization* which implies tolerance to system *transient faults*, i.e. faults that cause the state of the system (processes' local states and shared variables) to change arbitrarily. In that paper they present two wait-free clock synchronization algorithms for n processors which assume a global clock pulse ("in-phase" systems) and non-global 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 work on the same problem. By pointing out a simple approach in analyzing its difficulties, we show how to "re-parameterize" the $O(n^3)$ algorithm of [5], 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 computation model

The system consists of n identical processors. A processor p_i is a (possibly infinite) state machine. The processors communicate via a set of single-writer, multi-reader atomic shared variables. Each variable is owned by one processor. The owner of a variable can write it, while all the other processors can read it. Part of the state of each p_i is a pointer to the variables of some other processors in the system. In each one of its steps, p_i (i) reads the variables of the processor indicated by its current pointer value, (ii) changes state and (iii) updates its own variables. It must be noted that p_i has to read its own variables at each step because, as proven in [5], there can be no wait-free, self-stabilizing clock synchronization algorithm with only *blind* write operations (i.e. updates of shared variables without knowledge of their previous values).

We consider "in-phase" systems, in which all processors share a common clock pulse. Each pulse is a (possibly empty) set of processor names, which is 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 is said that it *missed the pulse.*

A *configuration* is a tuple of the processors' states and of the values of the shared variables. A system *execution* $\mathcal{E}$ is a sequence $c_0\pi_1c_1\pi_2\ldots$ of alternating pulses (denoted by π_x) and configurations (denoted by c_x). Each configuration c_i in a system execution is derived from its directly preceding one c_{i-1} by the state transitions and the shared variable updates of the processors that make a step in pulse π_i. The shared variable reads by all the processors that make a step in π_i return the respective values in c_{i-1}. 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. We say that a processor p_i makes l *continuous steps* if it makes steps for l *consecutive* pulses.

This system model, from the theoretical point of view, can be seen as describing the well-known theoretical PRAM model (cf. [10, 12]) with faults. In the real world it essentially describes existing synchronous multiprocessor systems (cf. [9]), in which faults may occur, or processors are scheduled independently. Pause intervals can be interpreted as faults in the connections of the pausing processor or as transient faults, or even as processor crashes.

In a solution to the clock synchronization problem, each processor owns a shared variable which encodes 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 in any execution $\mathcal{E}$ of the protocol the following conditions are satisfied:

•*Adjustment:* For any $l > k$ and for any processor p_i that makes l continuous steps during a sequence of l 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 two processors p_i and p_j and any sequence of $l \geq k$ consecutive pulses $\pi_{j+1}, \ldots, \pi_{j+l}$, in which both p_i and p_j have made l continuous steps, p_i's and p_j's clocks in c_{j+l} are equal.

For *self-stabilization* to be guaranteed by the solution the above two requirements should be met even in non-initialized executions. This suffices because a sub-execution that starts after transient faults have ceased can be viewed as a non-initialized execution.

3 Protocol Sync

The solution to the the clock synchronization problem that we present here—which is shown in pseudo-code in Figure 1—is based on the following strategy: each processor p_i (which has possibly missed some pulses) tries to catch up with the maximal clock in the system, by scanning in cyclic order the other processors' clocks and updating its own one to the maximum value it knows in each step, incremented by one. There is, however, a difficulty: the maximal clock in the system can remain hidden from p_i arbitrarily long, because the processors which hold and increment this maximal value may miss pulses just before being checked by p_i. Such a "game" may have unbounded duration (cf. [5]); moreover, if at any time point it stops, p_i will be likely to violate the adjustment requirement.

Since the problem is due to processors that misbehave by interchangeably switching between incrementing the maximal clock during some pulses and stopping operation in subsequent ones, the solution aims at preventing these processors from misleading the others that correctly and continuously work. Namely, when a processor realizes that it missed some pulse(s), it suspends its operation by not incrementing its clock for a certain number of its steps. Each p_i can detect whether it had stopped executing for some pulse(s), by counting, using its local array *prev* and the CNT_j shared variables, the number of steps that each p_j made since that last time p_i checked it.

In the approach taken in [5] the idea was that a continuously working processor, in order to catch up with the maximal clock in the system, needs $2(n-1)$ pulses during which no

```
shared var (CLOCK_1, CNT_1), ..., (CLOCK_n, CNT_n): (int, int) ;

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

Figure 1: Protocol SYNC for processor p_i

processor which increments its clock (i.e. not suspended) misses a pulse. Taking into account that in the worst case a processor might need $2n-3$ of its own steps to realize that it had missed a pulse and that there may be $n-1$ processors that try to mislead a correctly and continuously working one, that approach implied a suspension time of at least $2(n-1)^2(2n-3)$ steps, and, hence, a synchronization time of roughly $8n^3$ continuous steps (pulses).

Here we take a new approach, which improves the synchronization time by a factor of n. Consider a processor p_i that has taken some pause and its clock needs adjustment. After the end of its suspension period, if it correctly keeps making continuous steps, it is guaranteed that after it has performed a complete scan of the other processors' variables ($n-1$ steps) its own clock value will be no less than $n-1$ units smaller than the maximal clock value of the system at that time. During the $2n(n-1)$ pulses following that point, if the suspension period is $2n(n-1)$ steps long for each processor, there will be either (i) $n-1$ consecutive pulses in which a processor with the maximal clock value continuously makes steps, or (ii) (by the pigeon-hole principle) $n-1$ pulses, not necessarily consecutive, in which the maximal clock value is not incremented. Both these cases are convenient for p_i because it will either (i) actually read the maximal clock value in one of those steps, or (ii) have enough time to catch up with that value, respectively. Once it has the maximal clock value, p_i will continue holding the maximal clock value for as long as it keeps making continuous steps, since it will increment its clock by one at each step.

4 Analysis of the protocol

First we introduce some auxiliary terminology to simplify the presentation of the arguments.

- A processor p_i is *suspended* in a configuration c if its local variable $susp \neq 0$ in c.

- If c denotes a system configuration then

  (i) $CLOCK_i(c)$ denotes the value of the respective shared variable ($CLOCK_i$) in c,

  (ii) $MAX\_CLOCK(c) = max\{CLOCK_i(c) : 1 \leq i \leq n\}$, and

  (iii) $d_i(c) = MAX\_CLOCK(c) - CLOCK_i(c)$.

- A processor p_i performs a *forwarding step* in a pulse π_j if

  (i) p_i makes a step in π_j and

  (ii) $CLOCK_i(c_j) = MAX\_CLOCK(c_j)$ and

  (iii) $MAX\_CLOCK(c_j) = MAX\_CLOCK(c_{j-1}) + 1$.

  A pulse π_j is called *forwarding* if there exists some p_i which makes a forwarding step in π_j; otherwise it is called *non-forwarding* (in which case it is $MAX\_CLOCK(c_j) = MAX\_CLOCK(c_{j-1})$).

¿From now on, let $\mathcal{E}$ be a system execution (arbitrarily initialized) and let $\mathcal{E}_s$ be a sub-execution of $\mathcal{E}$ of length at least $k = (4n+1)(n-1)$ pulses and p_i be some processor that takes a step at each one of its pulses. We will prove that p_i, at most by the k-th of these steps, will hold the maximal clock value in the system. Let c_0 and c_4 denote the first and the last configurations of $\mathcal{E}_s$, respectively; let also c_1 be the configuration after the $(n-1)$-th pulse of $\mathcal{E}_s$, c_2 be the configuration after the $2n(n+1)$-th pulse of $\mathcal{E}_s$ after c_1 and, finally, c_3 be the configuration of $\mathcal{E}_s$ after the $(n-1)$-th pulse after c_2.

Lemma 4.1 *In any configuration c of $\mathcal{E}_s$ after configuration c_1 it will be $df \leq n-1$, where df is p_i's local variable.*

Proof. In its first $n-1$ steps in $\mathcal{E}_s$, p_i will load its array *prev* with the value of the CNT_x shared variable of every other processor p_x. ¿From that time on, since p_i is not missing pulses, it is going to calculate in *df* the number of steps that each p_x has done during the last period of $n-1$ pulses, during which p_i is taking continuous steps. □

Lemma 4.2 *In any configuration c of $\mathcal{E}_s$ after configuration c_2, p_i's local variable susp will equal* 0.

Proof. ¿From the previous lemma we have that after c_1, p_i will be finding $df \leq n-1$, and, consequently, it will be decrementing the value of *susp* by one at each pulse—if $susp \neq 0$—and will never increment it. Therefore, by the $2n(n-1)$-th pulse following c_1, p_i's local variable *susp* will equal 0. □

Lemma 4.3 *In configuration c_3 of $\mathcal{E}_s$ it will be $d_i(c_3) \leq n-1$. Moreover, for any two configurations c_j and c_{j+l} ($l \leq 2n(n-1)$) of $\mathcal{E}_s$ that occur after c_3 it will hold that $d_i(c_j) \geq d_i(c_{j+l}) + l_{nf}$, where l_{nf} is the number of non-forwarding pulses in the sub-execution specified by c_j and c_{j+l}.*

Proof. We first prove the first part of the lemma. Since at each step the maximal clock of the system can be incremented by at most one, it follows that:

$$MAX\_CLOCK(c_3) - MAX\_CLOCK(c_2) \leq n-1$$

But $MAX\_CLOCK(c_2)$ is the value of $CLOCK_x$ of some p_x in c_2, which p_i reads in one of these $n-1$ steps. Since $CLOCK$ variables are never decremented it follows that:

$$\begin{aligned} CLOCK_i(c_3) &\geq MAX\_CLOCK(c_2) \Rightarrow \\ MAX\_CLOCK(c_3) - CLOCK_i(c_3) &\leq MAX\_CLOCK(c_3) - MAX\_CLOCK(c_2) \end{aligned}$$

which, combined with the first inequality, implies that:

$$MAX\_CLOCK(c_3) - CLOCK_i(c_3) \leq n-1$$

The second part of the lemma can be derived by combining of the following two relations:

$$\begin{aligned} CLOCK_i(c_{j+l}) &\geq CLOCK_i(c_j) + l \\ MAX\_CLOCK(c_{j+l}) &= MAX\_CLOCK(c_j) + l - l_{nf} \end{aligned}$$

The former holds because p_i is not suspended (from lemma 4.2) and, thus, it increments its clock by at least one in each step. The latter holds because the system's maximal clock is incremented by one in each pulse, unless the pulse is non-forwarding. □

Lemma 4.4 *If during $\mathcal{E}_s$ and between configurations c_3 and c_4 there are $n-1$ or more non-forwarding pulses, then it will be $d_i(c_4) = 0$.*

Proof. It follows from Lemma 4.3 and from the following fact: if p_i at some step reads the maximal clock value of the respective configuration, then, as long as it works continuously it will keep holding the maximal clock value in the system and incrementing it (by incrementing its own clock) by one at each pulse. □

Lemma 4.5 *In configuration c_4 of $\mathcal{E}_s$ it will be $CLOCK_i(c_4) = MAX\_CLOCK(c_4)$.*

Proof. Assume, towards a contradiction, that $CLOCK_i(c_4) < MAX\_CLOCK(c_4)$. Let $\mathcal{E}_A$ denote the sub-execution specified by c_2 and c_4. Also, consider any processor p_x $(x \neq i)$ which makes steps during $\mathcal{E}_A$. We make two crucial remarks:
(i) Under our assumption, p_x cannot perform $n-1$ continuous forwarding steps during $\mathcal{E}_A$. 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 $\mathcal{E}_A$, it will load its local variable $prev[i]$ with a correct value of CNT_i written by p_i during $\mathcal{E}_A$; 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 $\mathcal{E}_A$: 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 sub-execution in which a processor is continuously suspended is at least equal to the length of $\mathcal{E}_A$ ($2n(n-1)$ pulses), p_x will not increment its clock again during $\mathcal{E}_A$.

What (ii) essentially implies is that the number of forwarding steps of each processor p_x $(x \neq i)$ in $\mathcal{E}_A$ is at most $2(n-1)$, which makes a total of at most $2(n-1)^2$ forwarding pulses in $\mathcal{E}_A$. The latter implies the existence of at least $2(n-1)$ non-forwarding pulses during $\mathcal{E}_A$, hence at least $2(n-1)$ ones after c_3. But then, by Lemma 4.4, p_i should hold the maximal clock value at c_4, which contradicts our assumption. □

Theorem 4.1 *Protocol* SYNC *is a self-stabilizing wait-free clock synchronization solution with* $k = (4n+1)(n-1)$.

Proof. After a processor p_i has worked continuously for $k = (4n+1)(n-1)$ steps, it is guaranteed by Lemma 4.5 that it will hold the maximal clock value in the system. After that, as long as it continues working correctly it will still hold the maximal clock value in the system and it will increment its clock by one at each pulse, thus satisfying the adjustment requirement. The same will hold with any other processor that has been working continuously and correctly for at least k pulses concurrently with p_i. This implies that its clock value will agree with the clock value of p_i, thus, the agreement requirement is satisfied, as well. The self-stabilizing property of the protocol is due to the fact that no initialization conditions were assumed for the analysis. □

Conclusions

In this work we show a wait-free and self-stabilizing protocol which achieves clock synchronization among n processors in at most $4n^2$ steps, and which improves the previously known solution which had synchronization time $O(n^3)$ steps. The best known non-stabilizing solution to the same problem has synchronization time $O(n^2)$, as well. Given these two facts, what deserves consideration is to study if the problem can be solved with a linear time algorithm or if the requirement for self-stabilization imposes some inherent overhead on the complexity of the problem; for a negative answer to the latter question, it suffices to prove that $O(n^2)$ is a lower bound for a wait-free even non-stabilizing solution to the problem.

Acknowledgments

We are thankful to Moti Yung for his help in the first steps of this work. We would also like to thank the anonymous referees of SWAT for their accurate and useful remarks. This work was done while the authors were visiting CWI, Amsterdam as guest researchers and were students of the Computer Engineering and Informatics Department, Patras University, Greece and CTI, Patras. The work was partially supported by NWO through NFI Project ALADDIN under contract number NF 62-376 and ALCOM ESPRIT Project Nr. 7141.

References

[1] A. Arora, S. Dolev, and M. Gouda. Maintaining Digital Clocks in Step. *Parallel Processing Letters 1*, 1, 1991, pp. 11-18.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

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

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

[10] 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.

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

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

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

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

[15] 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.

[16] T.K. Srikanth and S. Toueg. Optimal Clock Synchronization. *Journal of the ACM 34,* 3, 1987, pp. 626–645.

[17] M. Schneider Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[18] J.L. Welch and N. Lynch. A New Fault-Tolerant Algorithm for Clock Synchronization. *Information and Computation 77,* 1, 1988, pp. 1–36.