Parallel Processing Letters,
© World Scientific Publishing Company

ON SELF-STABILIZING WAIT-FREE CLOCK SYNCHRONIZATION

MARINA PAPATRIANTAFILOU and PHILIPPAS TSIGAS
Max Planck Institut für Informatik, Im Stadtwald
D-66123 Saarbrücken, Germany
Email: {ptrianta,tsigas}@mpi-sb.mpg.de

Received February 1996
Revised May 1997
Communicated by Michel Cosnard

ABSTRACT

A protocol which can tolerate any number of processors failing by ceasing operation for unbounded time and resuming operation (with or) without knowing that they were faulty is called *wait-free*; if it also works correctly even when the starting state of the system is arbitrary, it is called *wait-free, self-stabilizing*. This work is on the problem of wait-free, self-stabilizing clock synchronization of n processors in "in-phase" multiprocessor systems 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.

Keywords: Concurrency, Fault Tolerance, Logical Clocks, Multiprocessor Systems, Self-Stabilization, Wait-Free Synchronization

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 failures, as well as to design such fault-tolerant protocols for them.

In the past clock synchronization solutions that can tolerate failures have been proposed for the case of arbitrary, or Byzantine faults ([1,2,3] are just a few examples). In those models it has been proven that it is impossible to have a solution if the fraction of the non-faulty processors is less than one third [1] of the total. In the case of authenticated Byzantine faults the situation is not so bad; there exist algorithms that can tolerate any number of faults [4]. The negative results in that model are that the faulty processors can influence the clocks of the non-faulty ones by speeding them up, and that re-accession of repaired processors is not possible unless more than half of the processors are non-faulty [4]. *Self-stabilizing* algorithms for the clock synchronization problem have also been proposed [5,6]. 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 converges into a consistent global state and solves the task. For an introduction and a survey on self-stabilization, see [7,8].

To sum it all up, the "ideal" clock synchronization algorithm, that is highly resilient to failures, would have the following features:

(i) it would not only tolerate any number of processors' *napping faults* like in the authenticated Byzantine model, but also guarantee that the non-faulty processors' clocks remain unaffected by the failures;
(ii) it would allow faulty processors to rejoin the system when they resume normal operation and to become synchronized in a number of steps k (*synchronization time*) independent of the number of the working processors;
(iii) it would function correctly regardless of the system state in which it is started.

Recently, Dolev and Welch presented this highly resilient view of the problem as *wait-free, self-stabilizing clock synchronization* [9]; the first two aforementioned conditions capture the spirit of the *wait-freedom* [10,11] which implies maximum resilience to processor *halt/napping failures* and the third condition captures the spirit of *self-stabilization* which implies tolerance to system *transient faults* that cause the state of the system to change arbitrarily. They presented two wait-free clock synchronization algorithms for *in-phase* multiprocessors, achieving synchronization within $O(n^3)$ and $O(n^2)$ steps, respectively; the former solution is also *self-stabilizing*, while the latter depends on the initialization.

In this paper, by pointing out a simple approach in analyzing the difficulties of the problem we show how to "re-parameterize" the $O(n^3)$ algorithm of [9], to derive a clock synchronization protocol for in-phase multiprocessors —protocol SYNC, described in section 3— which is both *wait-free* and *self-stabilizing*, and achieves synchronization time $(4n + 1)(n - 1)$. The analysis and proof of correctness of the protocol, given in section 4, are simple and intuitive. Before proceeding with the presentation of the solution, in the next section we describe in more detail the system model.

2. The system model

The system consists of n identical processors, each of which 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, which is the only one that can write it, while all the others can read it. Part of the state of each processor p_i $(0 \leq i \leq n)$ is a pointer to the variables of some other processor. In each one of its steps, p_i (i) reads the variables of the processor pointed to 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, since, it has been proven, 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 [9].

We consider *in-phase* systems, in which processors share a common clock pulse. Each pulse is a (possibly empty) set of processor names, exactly those which *make a step* in the pulse. Each processor can make at most one step in one pulse. For a processor that does not make a step in a certain pulse we say that it *missed* that pulse. A *configuration* is a tuple of the processor states and 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 an 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 π_i. The shared variable reads in π_i return the respective values of c_{i-1}. An execution is *initialized* if its first configuration is explicitly specified by the protocol. We refer to a sub-sequence (starting and ending with a configuration) of a system execution $\mathcal{E}$ as a *sub-execution* of $\mathcal{E}$. We say that a processor p_i makes l *continuous steps* if it makes steps for l *consecutive* pulses.

This system model is an enhanced-with-failures version of the Parallel Random Access Machine model (cf. [12,13]). It describes common-clock-pulse multiprocessors (cf. [14]), in which faults may occur, or processors are scheduled independently; pause intervals also model faults in the connections of the pausing processor or transient faults, or even processor crashes. It is an important system architecture model that has been also shown to support universal wait-free consensus [15], which is not a trivial task and is a key to the power of the system [10].

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 $\mathcal{E}$ the following conditions are satisfied:

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

For *self-stabilization* to be guaranteed, the above two requirements should be met even in non-initialized executions; this suffices because a sub-execution starting after transient faults cease can be viewed as a non-initialized execution.

3. Protocol SYNC

Our wait-free self-stabilizing clock synchronization protocol —shown in pseudo-code in Fig. 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

```
shared var (CLOCK_1, CNT_1), ..., (CLOCK_n, CNT_n): (int, int) ;
protocol SYNC(i)
  var j, clock_j, cnt_j, df, my_clock, my_cnt, susp, prev[1..n]: int ;
  repeat
    for j = 1 to n (j ≠ i) do
        (clock_j, cnt_j) := read(CLOCK_j, CNT_j) ;
        my_cnt := my_cnt + 1 ; 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
```

Fig. 1. Protocol SYNC for processor p_i

clock to the maximum value it knows in each step, incremented by one. The difficulty is the following: 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. [9]), thus violating the requirements for adjustment and agreement in bounded time. 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 [9] 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 processor that increments its clock (i.e. that is not suspended) misses a pulse. Taking into account that in the worst case a processor might need $2n-3$ 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. During the pulses following its suspension period, if p_i correctly keeps making continuous steps, it is guaranteed that after it has performed a complete scan of the other processors' variables (this takes $n-1$ 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 p_i has the maximal clock value, it will continue holding it for as long as it keeps making continuous steps, since it will increment its clock by one at each step.

4. Analysis of protocol SYNC

We introduce some auxiliary terminology to simplify the presentation.

- A processor p_i is called *suspended* in a configuration c if its local variable *susp* has a non-zero value in c.
- If c is a system configuration then $\text{CLOCK}_i(c)$ is the value of CLOCK_i in c, $\text{MAX_CLOCK}(c) = max\{\text{CLOCK}_i(c) : 1 \leq i \leq n\}$, and $d_i(c) = \text{MAX_CLOCK}(c) - \text{CLOCK}_i(c)$ —i.e. $d_i(c)$ is the "distance" between the clock of p_i and the maximal clock of the system in c.
- Processor p_i makes a *forwarding step* in a pulse π_j if it makes a step in π_j and $\text{CLOCK}_i(c_j) = \text{MAX_CLOCK}(c_j)$ and $\text{MAX_CLOCK}(c_j) = \text{MAX_CLOCK}(c_{j-1}) + 1$.
- 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 $\text{MAX_CLOCK}(c_j) = \text{MAX_CLOCK}(c_{j-1})$).

Let $\mathcal{E}$ be an arbitrarily initialized execution and $\mathcal{E}_s$ be a sub-execution of $\mathcal{E}$ of length $k = (4n+1)(n-1)$ pulses and p_i be some processor that makes steps in all pulses in $\mathcal{E}_s$. We will prove that by the end of $\mathcal{E}_s$, p_i will hold the maximal clock value in the system. Let c_0 and c_4 be 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 following the $2n(n-1)$-th pulse after c_1 and c_3 be the configuration following the $(n-1)$-th pulse after c_2.

Lemma 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 loads 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 will calculate in df the exact number of steps that each p_x makes during every interval of $n-1$ pulses. □.

Lemma 2 *In any configuration c of $\mathcal{E}_s$ after configuration c_2, p_i's local variable susp equals zero.*

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 zero. □.

Lemma 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} 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:

$$\text{MAX_CLOCK}(c_3) - \text{MAX_CLOCK}(c_2) \quad \leq \quad n-1$$

But $\text{MAX_CLOCK}(c_2)$ 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:

$$\begin{aligned} \text{CLOCK}_i(c_3) \quad &\geq \quad \text{MAX_CLOCK}(c_2) \Rightarrow \\ \text{MAX_CLOCK}(c_3) - \text{CLOCK}_i(c_3) \quad &\leq \quad \text{MAX_CLOCK}(c_3) - \text{MAX_CLOCK}(c_2) \end{aligned}$$

which, combined with the first inequality, implies that:

$$d_i(c_3) \;=\; \text{MAX_CLOCK}(c_3) - \text{CLOCK}_i(c_3) \quad \leq \quad n-1$$

The second part of the lemma is derived by combining the two inequalities below:

$$\begin{aligned} \text{CLOCK}_i(c_{j+l}) \quad &\geq \quad \text{CLOCK}_i(c_j) + l \\ \text{MAX_CLOCK}(c_{j+l}) \quad &= \quad \text{MAX_CLOCK}(c_j) + l - l_{nf} \end{aligned}$$

The former holds because p_i is not suspended (from lemma 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 *If between configurations c_3 and c_4 in $\mathcal{E}_s$ there are at least $n-1$ non-forwarding pulses, then it will be $d_i(c_4) = 0$.*

Proof. The lemma follows from Lemma 3 and from the following fact: if p_i at some step reads the maximal clock value of that time, 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 5 *In configuration c_4 of $\mathcal{E}_s$ it will be $\text{CLOCK}_i(c_4) = \text{MAX_CLOCK}(c_4)$.*

Proof. Assume, towards a contradiction, that $\text{CLOCK}_i(c_4) < \text{MAX_CLOCK}(c_4)$. 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 $\mathcal{E}_A$.

Suppose that p_x performs $n-1$ *continuous forwarding* steps during $\mathcal{E}_A$. Since CLOCK_x is read by p_i every $n-1$ steps and p_i's steps in the specified interval are continuous by assumption, 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. Combining this with lemma 4 derives a contradiction.

Then the case that is left to be considered is the one that *no* p_x $(x \neq i)$ performs $n-1$ *continuous forwarding* steps during $\mathcal{E}_A$: Once p_x makes its first $n-1$ steps (not necessarily continuous) in $\mathcal{E}_A$, it is guaranteed that its variable $prev[i]$ contains a correct value of CNT_i —i.e. one 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, p_x cannot make more than $n-1$ forwarding steps in $\mathcal{E}_A$. This is so because if it does, we know from the previous case 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 $2n(n-1)$ pulses, which implies that p_x will not increment its clock again during $\mathcal{E}_A$. What the above reasoning essentially implies is that the number of forwarding steps of each processor p_x $(x \neq i)$ in $\mathcal{E}_A$ can be 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 $(n-1)$ ones after c_3. But then, by Lemma 4, p_i should hold the maximal clock value at c_4, which contradicts our assumption. □.

Theorem 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 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 for at least k pulses concurrently with p_i, hence its clock value will agree with the clock value of p_i, and the agreement requirement is satisfied. The self-stabilizing property of the protocol is due to the fact that no initialization conditions were assumed for the analysis. □.

Conclusions

We have shown a wait-free and self-stabilizing protocol for in-phase multiprocessors, 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 [9]. Given these two facts and the importance of the system model [15], what deserves consideration is to study if linear-time synchronization is achievable or if the requirement for self-stabilization imposes an inherent overhead on the complexity.

Acknowledgments

This work was done while the authors were visting CWI, Amsterdam as guest

researchers and were students of the Computer Engineering and Informatics Department, Patras University, Greece. It has been partially supported by NWO through NFI Project ALADDIN under contract number NF 62-376 and ALCOM ESPRIT Project Nr. 7141. A preliminary version appeared in the Proceedings of SWAT'94, LNCS Vol. 824, Springer-Verlag, 1994. We would like to thank the anonymous referees of SWAT for their accurate and useful remarks. We are also thankful to Moti Yung for his help in the first steps of this work.

References

[1] D. Dolev, J.Y. Halpern and H.R. Strong, On the Possibility and Impossibility of Achieving Clock Synchronization, *Journal of Computer Systems Science* **32** (1986) 230–250.

[2] S. Mahaney and F. Schneider, Inexact Agreement: Accuracy, Precision and Graceful Degradation, in *Proc. 4th ACM Symposium on Principles of Distributed Computing*, Aug. 1985, 237–249.

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

[4] J. Halpern, B. Simons, R. Strong And D. Dolev, Fault-Tolerant Clock Synchronization, in *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, Aug. 1984, 89–102.

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

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

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

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

[9] S. Dolev and J.L. Welch, Wait-Free Clock Synchronization, in *Proc. 12th ACM Symposium on Principles of Distributed Computing*, Aug. 1993, 97–108.

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

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

[12] R. Karp and V. Ramachandran, Parallel Algorithms for Shared Memory Machines, in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, ed. J.van Leeuwen, (Elsevier, Amsterdam 1990).

[13] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* (Morgan Kaufmann, 1992).

[14] K. Hwang, *Advanced Computer Architectures, Parallelism, Scalability, Programmability* (McGraw-Hill, 1993).

[15] M. Papatriantafilou and Ph. Tsigas, Wait-Free Consensus in *In-Phase* Multiprocessor Systems, in *Proc. 7th IEEE Symposium on Parallel and Distributed Processing*, Oct. 1995, 312-319.