A Time Optimal Self-Stabilizing Synchronizer Using A Phase Clock[*]

Baruch Awerbuch [†] Shay Kutten [‡] Yishay Mansour [§] Boaz Patt-Shamir [¶]
George Varghese [‖]

Abstract

A *synchronizer with a phase counter* (sometimes called asynchronous phase clock) is an asynchronous distributed algorithm, where each node maintains a local 'pulse counter' that simulates the global clock in a synchronous network. In this paper we present a time optimal self-stabilizing scheme for such a synchronizer, assuming unbounded counters. We give a simple rule by which each node can compute its pulse number as a function of its neighbors' pulse numbers. We also show that some of the popular correction functions for phase clock synchronization are not self-stabilizing in asynchronous networks. Using our rule, the counters stabilize in time bounded by the diameter of the network, without invoking global operations. We argue that the use of unbounded counters can be justified by the availability of memory for counters that are large enough to be practically unbounded, and by the existence of reset protocols that can be used to restart the counters in some rare cases where faults will make this necessary.

Index Terms: C. Computer Systems Organization C.2 COMPUTER-COMMUNICATION NETWORKS C.2.1 Network Architecture and Design Subjects: Distributed networks; G. Mathematics of Computing G.2 DISCRETE MATHEMATICS G.2.2 Graph Theory; General Terms: Algorithms; Reliability, Theory.

1 Introduction

A synchronizer [4] is a distributed algorithm that enables algorithms designed for synchronous networks to execute correctly on asynchronous networks. It simulates the "lock-step" property of synchronous networks. That is, in synchronous networks, the computation is performed in *rounds*, each performed exactly at the same time at all the nodes. The duration of a round is enough for a node v to receive messages from all the neighbors (or to *read* all the neighbors), to compute everything v needs to compute based on these values (and based on v's own state), and to send messages with the results to all the neighbors. (Those messages can be read by the neighbors in the next round.) Let *pulse* i at

[*] A preliminary version of this paper was included in an extended abstract that was presented in ACM STOC 1993.

[†] Department of Computer Science, Johns Hopkins University. baruch@cs.jhu.edu.

[‡] Department of Industrial Engineering and Management, Technion. kutten@ie.technion.ac.il.

[§] Department of Computer Science, Tel Aviv University mansour@math.tau.ac.il.

[¶] Department of Electrical Engineering, Tel Aviv University. boaz@eng.tau.ac.il.

[‖] Department of Computer Science, University of California, San Diego. varghese@cs.ucsd.edu.

Digital Object Indentifier 10.1109/TDSC.2007.1007 1545-5971/$25.00 © 2007 IEEE

node v be the event that starts the i-th round of the execution in node v. When no confusion arises we use the names round and pulse interchangeably.

A *synchronizer* is an algorithm that generate a series of events called *pulses* in every node in asynchronous networks. Pulse $i+1$ is generated in a node v after all the messages sent to node v in pulse i arrived. That is, the i-th pulse at node v can be used by an application at node v to know that the application can now compute the same results it could have computed in a synchronous network after round i ended. Note that this does not necessarily mean that v uses a pulse counter, and even if v does, the value $i+1$ is not necessarily exposed to v's neighbors. Indeed, the synchronizer of [4] uses a counter modulo 3.

A *phase clock* [30, 43, 26, 31, 28] in a synchronous network is a distributed algorithm that maintains in every node a copy of an integer valued variable, the *counter*, or the *clock*, that is incremented at every pulse. The values of all the copies are supposed to be always the same. (In the literature, such clocks are said to move in *unison* [26, 31, 28].) As noted e.g. in [30], this task is not trivial even when given that the network is synchronous.

In this note, we present a synchronizer that also supplies a phase clock for the resulting synchronous network. In fact, we do not address these two tasks separately. The implementation of the phase clock (in the asynchronous network) is our method for implementing the synchronizer. That is, in our synchronizer every node does keep the pulse number in a counter (clock) that is shared with its neighbors. (This can be implemented in a message passing system by sending this value to the neighbors from time to time.) Here, this value serves for two purposes. First, when the clock value at node v is $i+1$, an application at node v can know that messages of round i sent to v by v's neighbors already arrived. Second, v's neighbors can know that v is done sending messages for round i, and is now sending messages of round $i+1$.

When the algorithm is applied to an asynchronous network, the clock values at a node v may still be larger by 1 than the value at its neighbor u. Still, a synchronous algorithm can be applied to the network, since in this case v waits (before taking any step intended for the next time unit) until u's value is incremented.

As noted e.g. in [30], in a synchronous network, if the clocks have the same value, no communication is needed, since they are incremented at the same rate. Actions of the algorithms are then needed only to (1) detect that the clocks are "out of phase" and (2) bring the clocks back to the correct situation where they are equal. In asynchronous networks, the algorithm needs communication also in order to increment the clock, even when the clocks are "in phase".

Our algorithm has the important feature that it starts operating correctly *regardless of its initial state.* That is, it is *self-stabilizing* [19]. In practice, this means that the algorithm adjusts itself automatically to any change in the network or any unpredictable fault of its components, so long as the faults stop for some sufficiently long period. For example, assume that two partitions of a network merge (or two Peer to Peer networks merge) and the clock values in each partition are very different than those in the other. The algorithm will bring them to a correct global state where no values gap exists.

Our algorithm is simple and easy to implement. The algorithm stabilizes in time proportional to the diameter of the network, which is optimal. As detailed below, this improves both the time complexity of previous algorithms and the time complexity of algorithms that were published after the original version of this paper.

Problem statement. We are given an asynchronous message passing network, and our objective is to implement a *distributed pulse service* at the nodes. The service must provide the node at all times with a *pulse number*, subject to the following conditions.

Synchronization: Any message sent at local pulse i is received at the other endpoint before local pulse $i+1$ (of the other endpoint).

Progress: There exist parameters Δ and ρ (that may depend on the network topology), such that for any time interval of length $t > \Delta$, the number of consecutive pulses generated at every node is at least $\rho \cdot (t - \Delta)$. The parameter Δ is called *network slack* and ρ is the *progress rate.*

A self-stabilizing protocol is required to satisfy these conditions only after a certain *stabilization time* has elapsed. More intuition and elaborate description of the desired properties from such a service are given in Section 3 below.

The concept of unbounded counters. By the definitions used in this paper, the value of the clock is supposed to grow unboundedly. As detailed below, under *related work*, effort was invested in the literature in the non-trivial task of bounding the clock value. Hence, in terms of the assumption on the memory, the result in this paper is weaker than in some other papers we reference. On the on the other hand, the result in the current paper is stronger in terms of the time complexity. Having said that, let us try to motivate the theoretical assumption of "unbounded counters".

First, let us note that the theory of computer science often assumes memory unbounded machines. For example, a Turing machine is assumed to have an infinite tape. A machine with only a finite tape would be a finite state automaton, that is usually considered too weak a model to represent a realistic computer. An informal motivation may be that a real machine (as opposed to a Turing machine) may not posses infinite memory, but it has "enough" memory for the specific instances of the problems it encounters. Another informal assumption mentioned sometimes is that if a machine exhausts its memory, then it will be given additional memory, either by having people upgrade it, or by accessing some remote reserves.

Similarly, it is not difficult to use a clock variable that is "virtually" unbounded. For example, if the clock has several hundreds of bits, the clock value will not reach the bound unless a fault tampers with the value and sets it very high. Hence, had we not required self stabilization, an unbounded clock would not have posed any problem whatsoever (except in the sense of a constant factor in the memory complexity; recall that in this paper we address the time complexity).

A self stabilizing algorithm must assume that a fault may occur. Here, our assumption of an unbounded clock prevents such a fault from bringing the clock close to a bound. We note that unbounded memory even for fault tolerant systems and unbounded clocks are often assumed in the literature. See, for example [27, 31, 32, 2]. In addition, papers that use memory that is $O(\log n)$ when n, the number of nodes, is not known and may change, may also be viewed as using unbounded memory.

We would nevertheless like to mention that there exists a tool in the literature that can be used to solve the case that a clock that is supposed to be "practically unbounded" does reach a bound. This is not necessarily an efficient mechanism. However, if the size of the clock is so large that it is "practically unbounded" then this happens rarely, so efficiency maybe somewhat less important.

Such a mechanism is a self-stabilizing *reset* protocol. Note that reset is beyond the scope of the current paper (that deals with unbounded counters) and is mentioned here only as a motivation. However, Multiple such protocols were suggested in the literature, see e.g. [3, 9, 7, 45, 51, 23, 50, 13, 21, 5] to name just a few. Moreover, it was noted in [1] that it is rather easy to translate a self-stabilizing spanning tree construction protocol into a reset protocol. As mentioned below, the important work of Spinelli and Gallager [48] can also be transformed into a reset protocol, if we ignore the complexity.

A general approach to bounding unbounded counters using reset appeared in [8]. Please see references therein for papers presenting algorithms using unbounded counters. Let us just sketch here the idea in that paper. A reset protocol can be triggered by any non-empty subset of the nodes. It is supposed to bring the network into a target state that is some predefined (legal) initial state. In the case of an asynchronous system, a protocol (including a reset protocol) cannot "act at the same time" in different nodes. Hence, the actual target state is one that is a legal successor of the above initial state. Moreover, the actual target state is "close" to the desired initial state in the sense that an asynchronous algorithm cannot distinguish between them. More formally, both the actual target state and the intended one can have the same snapshot. Hence, if the clock value at some node (or nodes) reaches the bound, the reset protocol resets the values at all the nodes to an intended target initial state of zero value. (Actually, the system is reset to an actual target state where some of the values already moved up from zero, but are not "too far", say polynomial in n.)

Now, having established that "unbounded" clocks can be used in practice (and even that they can be bounded if so desired), let us try to discuss where can their use be more desirable. First, note that at least in synchronous networks, there exists a trivial solution that minimizes the number of clock states. That is, the pulse assumed in the synchronous networks model is already a clock whose number of states is 1. This is good enough for some applications. For other applications, the phase clock problem in *synchronous* network can be viewed, actually, as "how to *increase* the number of clock states?" (The observation that the case of a higher number of clock values may be harder than the case of a smaller number, is also implied in the work of [36]).

In general, application requirements may sometimes call for a clock that is unbounded. Applications written for a model where a clock is assumed may include a statement such as "wait x time (increments of the clock value) and then take an action", where x depends on the specific instance of the input. It may even be the case that x is computed by the application at run time, and that no a-priori bound on x is known in advance. Looking at an application program and finding out before an execution what is the value of x may even be undecidable. A clock of some constant number of states may not be able to supply this service for such an x.

Of course, an application may be able to use a constant number of state clock to implement its own, say, x states clock. However, this implementation requires solving a version of the phase clock problem: given a clock with a states, design a clock with a given $b > a$ states.

Finally, the designers of a clock with a bounded number of steps are more likely to take an action that decreases the value of the clock (e.g. to zero out its value). Such an action is indeed taken in algorithms presented in some of the papers mentioned below. We expect a typical application using a phase clock to expect the value of the clock to be monotonically increasing. Such an application may face problems with decreases in the value of the clock.

Previous and subsequent work. The problem discussed here bears some resemblance to the problem of clock synchronization as addressed, for example, by the Internet Time Protocol [42]. As mentioned in [11], there are differences between the two problems. Because of the very rich literature on the latter problem, we cannot survey it here.

Synchronizers were the target of considerable research, e.g. [4, 12, 47, 10, 6]. The concept of self-stabilization was introduced by Dijkstra [19]. A few general "stabilizer" schemes that upgrade non-stabilizing protocols to be self-stabilizing have been proposed. The problem (termed there *unison*) of maintaining clocks in phase in synchronous networks was discussed in [26] for unbounded clocks. The algorithm was not self-stabilizing.

In [31], two self-stabilizing unbounded phase clock algorithms for *synchronous* networks were suggested. We note here that it is easy to see that one of these algorithm (an adaptation of the algorithm of [26]) does stabilize also in asynchronous networks. However, its stabilization time is a function of the initial difference between the clocks values. This difference is controlled by faults, and hence it may be arbitrarily large. The second algorithm presented in [31] uses a rule that, as shown in the current paper, does not to stabilize in asynchronous networks.

Several papers were devoted to bounding the size (in bits, or in states) of the clock. Two stabilizing bounded values clock protocols for *synchronous* networks are presented [30]. The first was intended only for tree networks. Its stabilization time is the diameter of the network. The second is a probabilistic protocol for general (synchronous) networks. A specific setting is given in [30] where the *expected* stabilization time is exponential.

Other papers dealing with special topologies were suggested, e.g. [39, 33, 34]. In [11] a bounded value self-stabilizing algorithm for *synchronous* networks is presented. The problem there is named *digital clocks*, and applications to chip designing are explained. The rule used there is shown here not to stabilize for asynchronous networks. In [17] the assumptions are both that the the network is synchronous and that a node can read in an atomic step the values of all its neighbors. The algorithm uses a rule that we show not to stabilize in asynchronous networks. (The main contribution in [17] is the reduction in the size of the clock.)

All the above papers deal with the synchronous case. In [28] a bounded value self-stabilizing algorithm for *asynchronous networks* is given. The stabilization time there too is a function of the initial difference between the clocks' values. Recall that this could be arbitrarily large. We note that the advantage of our algorithm compared to that of [28] is manifest in the case that no roll-over is necessary (we assume unbounded counters) or if roll-over (reset) events are rare, as in the case when a clock has hundreds of bits. In other cases, such as a deeply embedded, very low-power network, relatively lean clocks with few bits are quite useful, because clock values can be piggy-backed on messages with minimal cost. In these cases, roll-over might be a relatively frequent, and it can be handled gracefully by the algorithm of [28].

Two papers that appeared after the original version of this paper deal with bounded counters, *asynchronous* networks and self-stabilizing protocols. They improved the time of other papers mentioned above, however, this improvement relies on the assumption that certain graph properties are known. In [29], it is assumed that a bound on the diameter of the network is known, and the stabilization time depends on this bound. In contrast, for our algorithm the time is of the order of magnitude of the *actual* diameter. (The main contribution of [29] is the useful additional property of time adaptivity; that is, the stabilization time is proportional to the number of faults.) Recently, in [16], it is assumed that the algorithm has access to a known bound on the size of cycles with certain properties (such as the largest hole in the graph). The stabilization time depends on that bound. (The main contributions in [16] are the bound on the size of the counter and the demonstration that phase clocks are applicable to a wide range of applications.) In [16], in every case that a node notices that its clock value is not consistent with that of a neighbor, the node in effect resets the whole system. We try to avoid such a drastic measure. For example, if a new node joins the network with its clock value being 0, the clock is corrected in $O(1)$ time without a reset.

Some of the clock papers deal with additional issues. For example, a node in some of the algorithms performs a step based on looking at one neighbor at a time. (In our algorithm a node first reads all the neighbors; however, we do not assume that this is done atomically). Other papers combine dealing with transient state faults (by self-stabilization) and dealing with other kinds of faults, [24, 25, 20, 46, 44]. The above survey is by no means complete. We mainly mention here papers that we also mention in the next section. There, we demonstrate why the logic used by some of these papers does not yield optimal stabilization time, or, sometimes, even stabilization at all, in the asynchronous model.

A somewhat different problem was studied in [35, 36, 37]. There, all the clocks should reach phase i before phase $i+1$ is started. (The implementation of a synchronizer, studied in the current paper, only requires that the difference between the clocks of neighbors is at most 1; this allows for large differences between clocks of nodes that are far away). In [35], a known upper bound on the diameter is assumed, and the stabilization time there depends on that upper bound, rather than on the actual diameter. In [36, 37] the stabilization time depends on the number of different clock values in the faulty state. In the worst case this number can be $\Omega(n)$ (where n is the number of the nodes), which is much larger than the actual diameter that is a constant in the system studied by [36]. (The main issue in [36, 37] is tolerating multiple types of faults.)

In the discussion of unbounded counters above we mentioned references for self stabilizing reset. It is worth stressing the important work of Spinelli and Gallager [48] on topological update in dynamic networks. This algorithm has many attractive features (in addition to its simplicity). The main property of the algorithm is that it stabilizes in time proportional to the diameter. Given such a self-stabilizing protocol, many tasks become much simpler, including the task of constructing a reset protocol. The space and communication complexity of this algorithm is high.

Our results. We present a simple new rule for self-stabilizing synchronization of networks, with network slack *diameter* and progress rate 1. Our rule does not invoke global operations, stabilizes in time linear in the actual diameter of the network without any prior knowledge, and does not require any additional memory space (other than for the pulse counter itself). By the lower bound of [39], this stabilization time is optimal. In the course of development of this rule, we obtain some interesting results regarding common synchronization rules, with applications to clock synchronization schemes. In particular, we show that rules used in other contexts (especially rules used in algorithms for synchronous networks) do not yield correct results here. We believe that the analysis of the new rule captures some of the inherent properties of synchronization.

Paper Organization. This paper is organized as follows. In Section 2 we define the basic notions and the model of computation we use. In Section 3, we develop the requirements from a desirable synchronization scheme by exploring the disadvantages of some popular schemes. In Section 4, we specify the new synchronization rule and analyze its complexity. In Section 5, applications to problems other than synchronizers are mentioned.

2 Notations and Model of Computation

We model the processor network as a fixed undirected graph $G = (V, E)$. For $u, v \in V$ we denote by $dist(u, v)$ the length of the shortest path between u and v. We follow the notational convention that $n = |V|$, and that $d = diameter = diameter(G) = \max_{u,v \in V} \{dist(u, v)\}$. For each node $v \in V$, we denote $\mathcal{N}(v) = \{u : dist(u, v) \leq 1\}$.

We assume the model of *unit capacity data links*, in which there is at most one outstanding message in transit on every channel at any given time. If an outgoing link is not empty, then the sender cannot send over it until the link becomes empty. (In [7, 50] it was justified as a realistic model for any message passing system with *some* bound on the capacity of the channels.)

The message delivery time can be arbitrary. For the purpose of analysis, we normalize time so that each message is delivered in at most one time unit. That is, given an execution (see, e.g. [41]) of a distributed algorithm assign any duration to the delivery time of each message, as long as this assignment is consistent with the execution (and the relation *happened-before* [38]). Call the longest such duration *one time unit.* The time complexity is the worst case (over the above duration assignments) number of time units of an execution. We note that computing the time according to this assumption is equivalent to computing it according to another common method of *asynchronous rounds*. See definition for time e.g. in [41, 49].

Note that by the time a message arrives at its destination, the state at the sender may have already changed. In terms the notion of the scheduler assumed sometimes in the context self stabilizing systems, this is equivalent to a *distributed daemon* [15]. Without loss of generality it is nevertheless common to assume in a distributed asynchronous system that an event (of receiving a message, computing, and possibly sending messages) at a node is atomic. Note that this does not restrict the model, since even if several nodes take an action exactly at the same time, the result of the actions of each node v cannot be known to other nodes until v's messages reach them.

We use the method of *local detection*, or *local checking* [1, 7, 50], in which each node constantly sends its state to all its neighbors. This enables us to present our protocol in a compact formulation of local *rules.* These rules are functions that take the state of the neighborhood (made available by the underlying local detection mechanism), and output a new state for the node.

3 Requirements and Examples

In this section, we consider a few preliminary ideas for synchronization rules. These ideas are not arbitrary. In addition to their being intuitive, these ideas were used in previous work, as well as in papers that appeared after the original version of this paper. In other contexts (mainly in the context of synchronous networks) these ideas were shown to yield correct algorithms, while in our context they fail (or, at least, are not time optimal). By studying the properties of these algorithms, we develop and demonstrate the requirements from a desirable scheme.

We start with a definition of the basic requirement of synchronization.

Definition 3.1 *Let $G = (V, E)$ be a graph, and let $P : V \to \mathbb{N}$ be a pulse assignment (where $\mathbb{N}$ is the set of natural numbers). We say that the configuration (G, P) is* legal for link (u, v), *denoted $legal(u, v)$, if $|P(u) - P(v)| \le 1$. The configuration is said to be* legal for a node v *if it is legal for all its incident links. The configuration is said to be* legal *if it is legal for all $e \in E$.*

The idea behind Definition 3.1 is as follows. In the synchronous setting, all messages sent at pulse i are received by pulse $i+1$. When we simulate executions of synchronous protocols on an asynchronous network, we do not have a global pulse-producing clock. Rather, we want to maintain the *validity* of the messages sent. This can be done by ensuring that a node sends pulse $i + 1$ messages only after it has **received** all the pulse i messages from its neighbors. Since message delivery is not simultaneous, there can be a skew of the pulse counters at neighbors, but this skew is allowed to be at most one: if the pulse numbers at two adjacent nodes differ by more than 1, then necessarily the node with the higher pulse number has advanced without receiving all messages of prior pulses. This notion of legal configuration gives rise to the following simple synchronization rule, which is implicit in the α synchronizer of [4]. (We note that self-stabilization is not a concern in [4]; however, similar rules are used in papers that did ensure self-stabilization [11, 31, 30, 17, 34].)

Rule 1 *(Min Plus One)*

$$P(v) \leftarrow \min_{u \in \mathcal{N}(v)} \{P(u)\} + 1$$

The idea is that just before the pulse number is changed, the node sends out all the messages of previous rounds which haven't been sent yet. Note that since $v \in \mathcal{N}(v)$, we have that $\min_{u \in \mathcal{N}(v)} \{P(u)\} \le P(v)$, and hence $P(v)$ cannot increase by more than one in a single application of Rule 1. This has the nice consequence that each node goes through all pulse numbers, as expected.

Stabilization issues. As is well known, Rule 1 is *stable*, i.e., if the configuration is legal (as in Definition 3.1), then applying the rule arbitrarily can yield only legal configuration. Notice however, that if the state is not legal, then applying Rule 1 may cause pulse numbers to drop. This is something to worry about, since the regular course of the algorithm requires pulse numbers only to grow. Thus, it is conceivable that actions taken in legal neighborhoods are adversely affected by the actions taken in illegal neighborhoods. This intuition is captured by the following theorem.

Theorem 3.1 *Rule 1 is not self-stabilizing.*

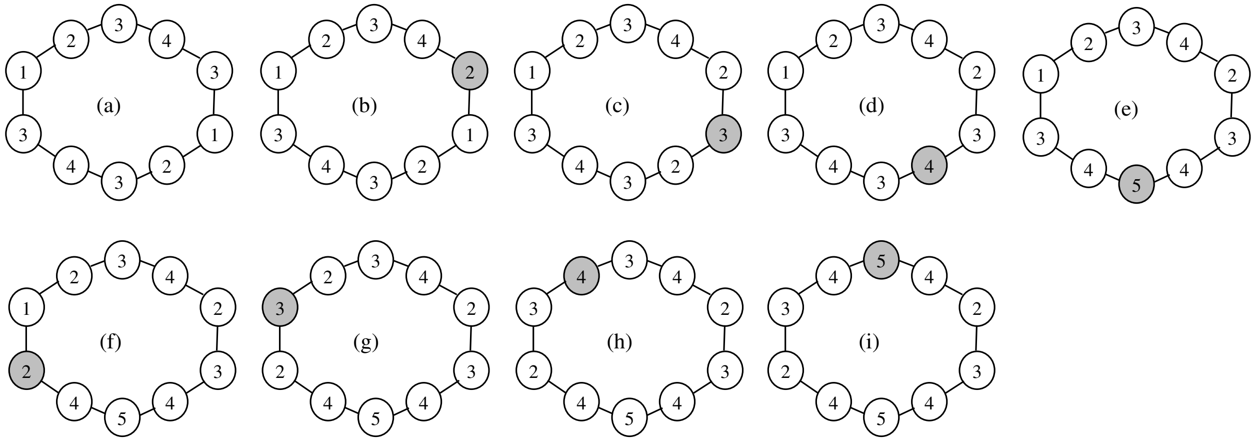

Figure 1: *An execution using Rule 1. The node that moved in each step is marked.*

Proof: By a counter example. Consider the pulse configuration of a 10-node ring depicted in Figure 1 (a). The vertical edges represent illegal links. Consider now the execution described in Figure 1 (a-i), obtained by the repeated application of Rule 1. It is readily seen that the last configuration (i) is basically identical to configuration (a), with all the pulse numbers incremented by one, and rotated one step counter-clockwise. Repeating this schedule results in an infinite execution in which each processor takes infinitely many steps, but none of the configurations is legal. ▮

Let us make a short digression here. The above leads also to an interesting observation for *clock synchronization*: one of the popular schemes for clock synchronization [40] is "repeated averaging". Roughly speaking, under the repeated averaging rule, each node sets its value to be the average value of its neighbors, while advancing the clock if this average is close enough to its own value.

Observation 3.2 *Repeated averaging does not stabilize.*

Proof Sketch: The scenario in the proof of Theorem 3.1 shows that averaging with rounding down does not work. A similar scenario can be constructed for averaging with rounding up.

Time Complexity issues. Consider Figure 1 (a). Note that the illegal state manifests in the fact that the nodes with the minimum clock values have value gaps with their neighbors (1 vs. 3). Intuitively, one would hope that the illegal state is resolved by the fact that the nodes with the minimum value increment their clocks, and catch up with the others. Looking at Figure 1 (i), the nodes with the minimum values indeed incremented their clocks. However, other nodes decreased the values of their clocks, and became the new minimums. Moreover, the illegal state in 1 (i) manifests again in the gap in the values between these minimum nodes and their neighbors (2 vs. 4).

One idea that can pop into mind in trying to repair the flaw exposed by the above proof is never to let pulse numbers go down. A similar rule appears in [28] (except for the case of a wrap-around of the bounded clock). In [31] (Section 5), such a rule is suggested as a modification that makes the rule of [26] self-stabilizing. Formally, the rule is the following:

Rule 2 *(Monotone Min Plus One)*

$$P(v) \leftarrow \max\left\{P(v), \min_{u\in\mathcal{N}(v)} \{P(u)\}+1\right\}$$

Rule 2 can be proven to be self-stabilizing. However, it suffers from a serious drawback regarding its stabilization time. Consider the configuration depicted in Figure 2.

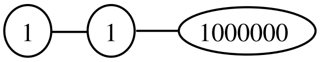

Figure 2: *A pulse assignment for Rule 2.*

A quick thought should suffice to convince the reader that the stabilization time for this configuration using Rule 2 is about 1000000 time units, which seems to be unsatisfactory for such a small network. This example demonstrates an important property that we shall require from any self-stabilizing protocol: *the stabilization time must not depend on the initial state; rather, it should be bounded by a function of the network topology.* A clear lower bound on the stabilization time is the diameter of the network (e.g., if $n-1$ nodes must change their pulse number). In the example above, using Rule 2, the stabilization time depends linearly on the value of the pulses, thus implying that the stabilization time can be arbitrarily large.

Asynchrony issues. The next idea is to have a combination of rules: certainly, if the neighborhood is legal, then the problem specification requires that Rule 1 is applied. However, if the neighborhood is not legal, another rule can be used. The first idea we consider is the following *Maximum* rule for the case that a node detects locally that the network is not stable. A similar rule is used in [31, 11, 20] in the context of synchronous networks.

Rule 3 *(Maximum)*

$$P(v) \leftarrow \begin{cases} \min_{u\in\mathcal{N}(v)} \{P(u)\}+1\ , & \textit{if legal}(v) \\ \max_{u\in\mathcal{N}(v)} \{P(u), P(v)\}\ , & \textit{otherwise} \end{cases}$$

It is straightforward to show that if an atomic action consists of a node reading its neighbors and setting its own value (in particular, no neighbor changes its value in the meanwhile), then Rule 3 above converges to a legal configuration. Unfortunately, this model, traditionally called the *central demon* [19, 14], requires tight synchronization between nodes, which isn't considered realistic usually. As shown in [31, 11], a similar rule suffices even without such an atomic action, but assuming a synchronous network. Unfortunately, Rule 3 does not work in asynchronous networks without a central demon.

Theorem 3.3 *Rule 3 is not self-stabilizing in an asynchronous system.*

Proof:

A scenario where the system does not self stabilize is given in Figure 3. The move from the configuration of part (c) of the figure to that of part (d) may seem surprising. However, in an asynchronous environment, the message that the left node increased its clock value from 2 to 4 (moving

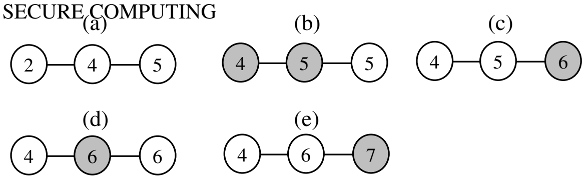

Figure 3: *An execution using Rule 3 in a truly distributed system. The nodes that send or receive are marked.*

from part (a) to part (b)) may be slower than messages from the node on the right. In this case, the estimation at the middle node of the value of the left node in configuration (c) is 2. The middle node then applies the correcting part of Rule 3, resulting in yet another illegal configuration (d). Since configuration (e) is equivalent to configuration (a) with pulse numbers incremented by 2, we conclude that repeating this schedule results in an execution with infinitely many illegal states. ∎

Locality and Simplicity issues: Finally, we would like to address two properties which are somewhat harder to capture formally: *locality and simplicity.* It seems, however, that these properties are of the highest importance in practice. The rules discussed above do exhibit these properties, even though each such rule had other disadvantages. In mending the above rules we should not spoil the locality and simplicity they exhibit.

By *locality* we mean that it is much preferable that a processor will be able to operate while introducing only the minimal possible interference with other nodes in the network. In other words, invoking a global operation is considered costly, and we would like to avoid it as much as possible. One way to capture this intuition approximately in our case is to require that the only state information at the nodes is the pulse number, and that protocols should operate by applying local rules as above.

As an illustrative exercise, contrast this approach with the following solution: whenever an illegal state is detected, *reset* the whole system. This solution, although it may be unavoidable in some cases, does not seem particularly appealing as a routinely activated procedure. Consider, for example, the common situation in which a new node joins the system (perhaps it was down for some while). We would like the protocols to feature *graceful joining* in this case, i.e., that other nodes would be affected only if necessary (e.g., the neighbors). Note that a rule that reduces the value of a clock always to the minimum among the neighbors will not handle such a join gracefully even if no reset is used. That is, the joining node may have a zero value for its pulse. A rule such as Rule 1 can cause all the nodes in the network to reduce the values of their clocks significantly.

The last property we require from distributed protocols is even harder to define precisely. Essentially, we would like to have the protocols conceptually *simple.* This will make the protocols easy to understand, and therefore, easy to implement and maintain. This requirement is one of the main obstacles for many sophisticated protocols that are not used in practice.

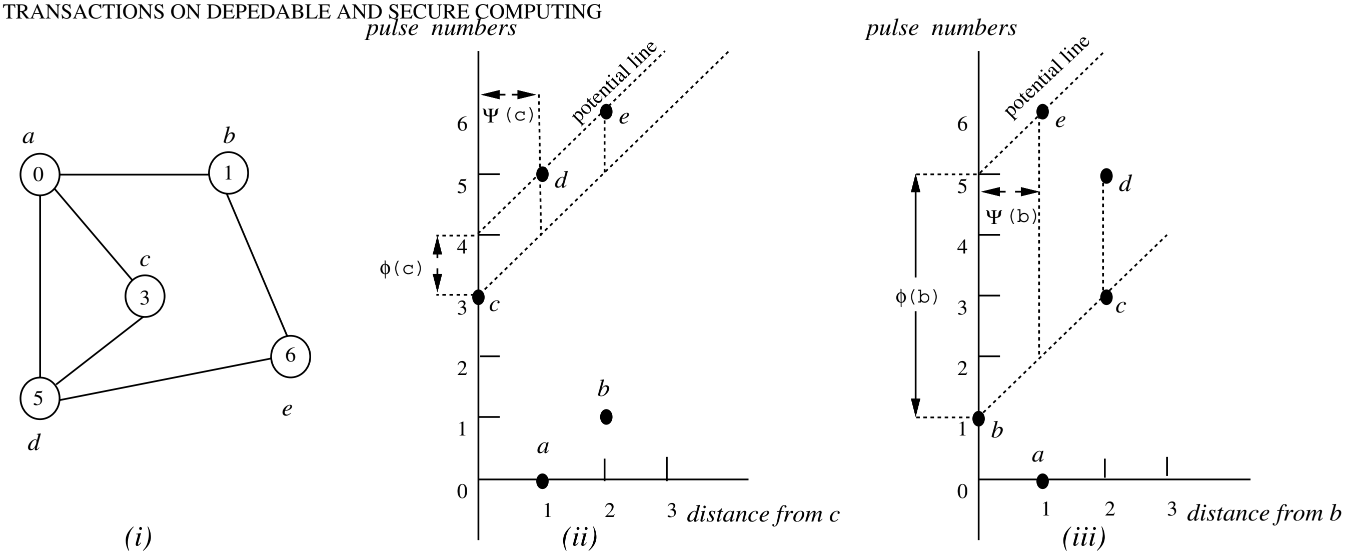

Figure 4: *On the left is an example of a graph with pulse assignment (i). Geometrical representations of this configuration are shown in (ii) and (iii). The plane corresponding to node c is shown in the middle (ii), and the plane corresponding to node b appears on the right (iii). As can be readily seen,* $\phi(c) = 1$*, and* $\phi(b) = 4$*. Also,* $\Psi(c) = 1$*, and* $\Psi(b) = 1$ *(see Definition 4.2).*

4 An Optimal Self-Stabilizing Rule

In this section, we give a simple, self-stabilizing, optimal rule of synchronization, and analyze its stabilization time. Specifically, our synchronization scheme is based on the following rule.

Rule 4 *(Max Minus One)*

$$P(v) \leftarrow \begin{cases} \min_{u \in \mathcal{N}(v)} \{P(u)\} + 1 \ , & \textit{if legal}(v) \\ \max_{u \in \mathcal{N}(v)} \{P(u) - 1, \ P(v)\} \ , & \textit{otherwise} \end{cases}$$

In words, Rule 4 says to apply a “minimum plus one” rule (Rule 1) when the neighborhood seems to be in a legal configuration, and if the neighborhood seems to be illegal, to apply a “maximum minus one” rule (but never decrease the pulse number). The similarity to the “maximum” rule (Rule 3) is obvious. The intuition behind the modification is that if nodes change their pulse numbers to be the *maximum* of their neighbors, then “race conditions” might evolve, where nodes with high pulse numbers can “run away” from nodes with low pulse numbers. Since the correction action takes the pulse number to be one less than the maximum, nodes with high pulse number are “locked,” in the sense that they cannot increment their pulse counters until all in their neighborhood have reached their pulse number. This “locking” spreads automatically in all the “infected” area of the network. Formally, the way Rule 4 corrects any initial state is analyzed in detail in the proof of Theorem 4.1 below.

Theorem 4.1 *Let* $G = (V, E)$ *be a graph with diameter d, and let* $P : V \rightarrow \mathbb{N}$ *be a pulse assignment. Then applying Rule 4 above results in a legal configuration in at most d time units.*

In order to prove Theorem 4.1, we develop some tools to analyze the behavior of the synchronization scheme. The basic concept that we use is a certain *potential* value we associate with every node, described in the following definition.

Definition 4.1 *Let v be a node in the graph. For any node u, the* wave height *of u over v is*

$$D_v(u) \;=\; P(u) - P(v) - dist(u,v) \;.$$

The potential *of v, denoted $\phi(v)$, is*

$$\phi(v) = \max_{u \in V} \{D_v(u)\} \;.$$

Intuitively, $D_v(u)$ measure by how much is the pulse number of v lagging behind the pulse number of u, after allowing for a correction due to the distance between them. The potential, $\phi(v)$, is a measure of the largest distance-adjusted skew in the synchronization of v. Graphically, one can think that every node u is a point on a plane where the x-coordinate represents the distance of u from v, and the y coordinate represents the pulse numbers (see Figure 4 for an example). In this representation, v is at the origin (and it is the only node on the line $x = 0$), $D_v(u)$ is the vertical distance between u and the 45 deg line from the origin, and $\phi(v)$ is the maximal vertical distance $D(u)$ of any point (i.e., node u) above the 45-degree line $y(x) = P(v) + x$.

Let us start with a few simple properties of ϕ.

Lemma 4.2 *For all nodes $v \in V$, $\phi(v) \geq 0$.*

Proof: By definition, $\phi(v) \geq D_v(v) = 0$. ▮

Lemma 4.3 *A configuration of the system is legal if and only if for all $v \in V$, $\phi(v) = 0$.*

Proof: Suppose first that $\phi(v) = 0$ for all v. Then, in particular, for each edge (u, v) we have that $P(u) - P(v) = D_v(u) + 1 \leq \phi(v) + 1 = 1$ and similarly $P(v) - P(u) \leq 1$, namely the difference between the pulse numbers of every pair of neighbors is at most 1, so the configuration is legal. Conversely, suppose now that $\phi(v) > 0$ for some node v. Then for some node u, $D_v(u) > 0$, i.e., $P(u) - P(v) > dist(u, v)$. Pick any path from u to v of length $dist(u, v)$: since the difference in the pulse assignments in the endpoints is larger than the number of links along the path, by the pigeonhole principle there must exist a link (w, y) on the path such that $P(y) - P(w) > 1$, and hence the configuration is illegal. ▮

So far, we analyzed a given configuration, in terms of values of the pulse counters at the nodes. We now turn to analyze the way configurations change dynamically over time. To this end, we must to consider the effect of the asynchronous schedule. In particular, at any given instant, the value of a pulse counter at a node may not necessarily be the its value as perceived by its neighbor. We denote the pulse value of a node x as perceived by its neighbor y by $P_x(y)$. Another notational convention we adopt is that when we talk about two states where one occurs before the other, we denote all functions of the later state by a prime, e.g., $\phi(v)$ is the potential of v in the first state and $\phi'(v)$ is the potential of v in the later state.

To facilitate the analysis, we first state the self-stabilization property we assume on the links.

Lemma 4.4 *Suppose that the computation starts in an arbitrary state, consider the sequence of pulse values $P_u(w)$ read by a node u from a neighbor w, and the sequence of $P(w)$ values in node w. Then with the possible exception of a prefix of at most T_0 time units, these sequences are identical, and moreover, every value of $P(w)$ appears as a value of $P_u(w)$ after at most one time unit.*

For the remainder of the discussion, we will identify the start of the computation with the aforementioned T_0, i.e., our analysis assumes that the links have already stabilized.

The following corollary is the key to dealing with an asynchronous schedule.

Lemma 4.5 *For all neighbors u, v, at all times, we have $P_v(u) \le P(u)$.*

Proof: Follows from Lemma 4.4 and the fact that the sequence of pulse numbers in a node is non-decreasing. ▮

Lemma 4.6 *Consider a given state, and suppose that some node u changes its pulse number by applying Rule 4. Then for all nodes $v \in V$, $\phi'(v) \le \phi(v)$.*

Proof: The first easy case to consider is the potential of u itself. Since by Lemma 4.5 $P'(u) > P(u)$ in this case, we have

$$\begin{aligned}
\phi'(u) &= \max_{w\in V}\left\{P'(w) - P'(u) - \mathit{dist}(w,u)\right\} \\
&= \max\left(\max_{w\neq u}\left\{P(w) - \mathit{dist}(w,u) - P'(u)\right\}\ ,\ 0\right) \\
&\le \max_{w\in V}\{P(w) - P(u) - \mathit{dist}(w,u)\} \\
&= \phi(u)\ .
\end{aligned}$$

Let us now consider $v \neq u$. The only value that was changed in the set $\{D_v(w) \mid w \in V\}$ is $D_v(u)$. We prove that $D'_v(u) \le D_v(w)$ for some $w \in \mathcal{N}(u)$, and hence $\phi'(v) \le \phi(v)$. There are two cases to consider. First, assume that u changed its pulse by applying the "min plus one" part of Rule 4. Since $u \neq v$, there must be a node $w \in \mathcal{N}(u)$ with $\mathit{dist}(w,v) = \mathit{dist}(u,v) - 1$. Also, since "min plus one" was applied, and using Lemma 4.5, we have $P'(u) \le P_u(w) + 1 \le P(w) + 1$. Therefore,

$$\begin{aligned}
D'_v(u) &= P'(u) - P(v) - \mathit{dist}(u,v) \\
&\le (P(w)+1) - P(v) - (\mathit{dist}(w,v)+1) \\
&= P(w) - P(v) - \mathit{dist}(w,v) \\
&= D_v(w)\ ,
\end{aligned}$$

and hence $\phi'(v) \le \phi(v)$ in this case.

The second case to consider is when u has changed its value by applying the "max minus one" part of Rule 4. The reasoning is dual to the first case: let w be a neighbor of u such that $P_u(w) = \max\{P_u(r) \mid r \in \mathcal{N}(u)\}$. Then, using Lemma 4.5, we have $P(w) \ge P_u(w) + 1 = P'(u) + 1$. By the triangle inequality, $\mathit{dist}(w,v) \le \mathit{dist}(u,v) + 1$, and hence

$$\begin{aligned}
D'_v(u) &= P'(u) - P(v) - \mathit{dist}(u,v) \\
&\le (P(w)-1) - P(v) - (\mathit{dist}(w,v)-1)
\end{aligned}$$

$$
\begin{aligned}
&= P(w) - P(v) - \mathit{dist}(w,v)\\
&= D_v(w)\ ,
\end{aligned}
$$

and we are done. ▌

In other words, each time a node with a positive potential changes its pulse number, its potential strictly decreases. This fact, when combined with Lemmas 4.2 and 4.3, can be used to prove eventual stabilization. However, this argument only shows that the stabilization time is bounded by the total potential of the configuration, which in turn depends on the initial pulse assignment. We need a stronger argument in order to prove a bound on the stabilization time that depends only on the topology, as asserted in the statement of Theorem 4.1. Toward this end, we define the notion of "wavefront."

Definition 4.2 *Let v be any node. The* wave *of v, denoted $W(v)$ is the set*

$$W(v) = \{u \in V \mid D_v(u) = \phi(v)\}\ .$$

The wavefront distance *of v, denoted $\Psi(v)$, is defined by*

$$\Psi(v) = \min\{\mathit{dist}(u,v) \mid u \in W(v)\}\ .$$

In the graphical representation, the wavefront distance of a node is simply the distance to the closest node on the "potential line" (see Figure 4 for an example). Intuitively, one can think of $\Psi(v)$ as the distance to the "closest clock that is maximum too far ahead" of v. The importance of the wavefront becomes apparent in Lemma 4.10 below, but let us first state an immediate property it has.

Lemma 4.7 *Let $v \in V$. Then $\Psi(v) = 0$ if and only if $\phi(v) = 0$.* ▌

The following lemma captures the "locking effect" of the "max minus one" part of Rule 4.

Lemma 4.8 *Suppose that at a given state, $\phi(v) > 0$ and $u \in W(v)$. Then applying Rule 4 does not change the pulse number at u.*

Proof: First we claim that for all $w \in \mathcal{N}(u)$ we have that $P(w) \le P(u) + 1$, because

$$P(w) = D_v(w) + P(v) + \mathit{dist}(v,w)\ \le\ D_v(u) + P(v) + \mathit{dist}(v,u) + 1 = P(u) + 1\ .$$

It follows that any application of the "max minus one" part of Rule 4 cannot increase the number at u. As for the "min plus one" part, suppose that all links incident to u are legal. Observe that $u \neq v$ because $\phi(v) > 0$. Let $w \in \mathcal{N}(u)$ be such that $\mathit{dist}(v,w) = \mathit{dist}(v,u) - 1$, i.e., w is on the shortest path from v to u. Since $u \in W(v)$, we have $D_v(w) \le D_v(u)$, and hence

$$P(w) = D_v(w) + P(v) + \mathit{dist}(v,w)\ \le\ D_v(u) + P(v) + \mathit{dist}(v,u) - 1 = P(u) - 1\ .$$

Therefore, if u applies the "min plus one" part of Rule 4, the resulting pulse number cannot be more than $P(w) + 1 = P(u)$, namely $P(u)$ remains unchanged. ▌

The following lemma says that $W(v)$ is monotonically growing, so long as the potential of v is positive.

Lemma 4.9 *Suppose that at a given state, $u \in W(v)$. If in the following state $\phi'(v) > 0$, then $u \in W'(v)$.*

Proof: Suppose that node x applied Rule 4 between the two states. By Lemma 4.8, if $x = u$ then the state remains unchanged and the lemma follows trivially. If $x = v$ then for all nodes $w \neq v$, $D'_v(w) = D_v(w) - (P'(v) - P(v))$, i.e., all D_v values are decreased by the same amount (which is exactly the increase in the pulse number of v). Therefore, $D'_v(u)$ remains maximal among $D'_v(w)$ values for $w \neq v$. if $\phi'(v) > 0$, then $D'_v(u) > D'_v(v)$ and hence $u \in W'(v)$. Finally, suppose that $x \notin \{u, v\}$ moved. In this case, $D'_v(u)$ does not change, and hence $\phi'(v) = \phi(v)$ by Lemma 4.6. It follows that $u \in W'(v)$ in this case too. ▮

Lemma 4.10 *Suppose that at a given state we have $\Psi(v) > 0$. Then after 1 time unit we have $\Psi'(v) \leq \Psi(v) - 1$.*

Proof: Suppose $\Psi(v) = f > 0$ at the given state. If after one time unit we have $\phi'(v) = 0$ we are done, since in this case $\Psi'(v) = 0$ by Lemma 4.7. So assume henceforth that $\phi'(v) > 0$. Consider any node $u \in W(v)$ such that $dist(u, v) = \Psi(v)$. Let $\text{closer}_v(u) \in \mathcal{N}(u)$ be such that $dist(v, \text{closer}_v(u)) = f - 1$, i.e., $\text{closer}_v(u)$ is on the shortest path from v to u. Note that such a node $\text{closer}_v(u)$ must exist because $\Psi(v) > 0$ and hence $u \neq v$. We claim that after 1 time unit, $P'(\text{closer}_v(u)) \geq P(u) - 1$. To see that, first note that at the given state, $P(\text{closer}_v(u)) \leq P(u) - 1$ because $\text{closer}_v(u) \notin W(v)$ (or otherwise $\Psi(v)$ would have been $f - 1$). This means that the link $(\text{closer}_v(u), u)$ is in an illegal state. Therefore, after at most one time unit, $\text{closer}_v(u)$ will read the pulse number of v, and by Rule 4 will set its pulse number to be at least $P(u) - 1$. It follows that after one time unit, for each node in $u \in W(v)$ there exists a node $\text{closer}_v(u)$ which is closer to v by one unit, and whose pulse number is at least $P(u) - 1$, which means that $D'_v(\text{closer}_v(u)) \geq D'_v(u)$. Since by Lemma 4.9, $u \in W'(v)$, we must also have $\text{closer}_v(u) \in W'(v)$, and therefore, $\Psi'(v) \leq \Psi(v) - 1$. ▮

The next corollary follows from Lemmas 4.7 and an inductive application of Lemma 4.10.

Corollary 4.11 *Let v be any node. Then after at most $\Psi(v)$ time units, $\phi(v) = 0$.*

We can now prove Theorem 4.1.

Proof of Theorem 4.1: . By Lemma 4.3, it suffices to show that after at most d time units, $\phi(v) = 0$ for all $v \in V$. From Corollary 4.11 above, we actually know that a slightly stronger fact holds: for all nodes $v \in V$, after at most $\Psi(v)$ time units, $\phi(v) = 0$. The theorem follows from the facts that for all $v \in V$, $\Psi(v) \leq d$, and by the fact that $\phi(v)$ never increases, by Lemma 4.6. ▮

It is straightforward to see that starting in a legal configuration, every application of Rule 4 leaves the system in a legal configuration, as long as no additional faults occur. Clearly, our Rule 4 satisfies the synchronization condition, namely, any message sent at local pulse i is delivered at the other endpoint before its local pulse $i + 1$ (see discussion after Definition 3.1). Finally, note that our synchronizer has progress rate 1 and network slack equal to the diameter of the network.

5 Conclusion

Many of the related papers deal with aspects that are not studied in the current paper. Below we mention some such issues and discuss how they may be relevant to our paper or to future research.

The tool proposed here seems to motivate the research about composition between self stabilizing protocols to obtain a good time complexity. Specifically, one may want to use a reset procedure together with the algorithm proposed here, in order to make the result work for bounded clocks. Such a specific kind of composition was discussed in [8], as well as in the original version of this paper, but only for the task at hand. This issue was later developed in a wider context in other papers, such as [22]. It seems that there are still interesting open questions in this area.

In this paper we devised a phase clock as a method to synchronize networks in a self-stabilizing manner. As noted in [16, 26, 20, 31, 11], a phase clock has many other applications. An interesting question is whether better algorithms exist for specific applications. In particular, should one need a synchronizer only (the application discussed in this paper). Does there exist better algorithms for that problem (better than for the asynchronous phase clock problem)? For example, multiple papers dealt with the issue of bounding the size of the phase clock. Given a lower bound on the size of phase clocks, does this lower bound apply to the problem of constructing a self-stabilizing synchronizer?

As mentioned above, a trivial solution of one state exists in *synchronous* networks. However, such a solution is not useful (beyond the service that a synchronous network provides anyhow). It was shown in [39] that no deterministic clock algorithm for a clock with a constant number of bits exists. It was mentioned in [11] that their scheme needs $\Omega(n)$ states in some graph in order not to deadlock. Note that any local scheme (incrementing the value of the clock based only on the value of the clocks of a node and its neighbors) must use Rule 1 in the case that the clock values are legal. Since this is the rule used in [11], there exist graphs where the size of the clock is bounded from below by $\Omega(n)$ for "local" schemes. (As shown in [16], there exist smaller clocks in many graphs.) Does this lower bound apply to local schemes for a self stabilizing synchronizer? What is the minimum size of a clock if the algorithm is not required to be local (in the sense mentioned above)? It seems likely that such a non-local algorithm may consume more time between pulses. Is there a time-rate trade-off? Moreover, our time optimal algorithm is not space optimal. Does there exist an algorithm that is optimal in all the parameters?

Several papers deal with models that allow additional kinds of faults, such as *crash* faults, *napping* faults, or *Byzantine faults*. See e.g. [24, 25, 20, 46]. One can ask what is the optimal stabilization time in these models? Is it inherently larger than the model of self-stabilization?

References

[1] Yehuda Afek, Shay Kutten, and Moti Yung. Local Detection for Global Self Stabilization. Theoretical Computer Science , Vol 186 No. 1-2, 339 pp. 199-230, October 1997.

[2] Sudhanshu Aggarwal, Shay Kutten. Time Optimal Self-Stabilizing Spanning Tree Algorithms. FSTTCS 1993: 400-410.

[3] Anish Arora and Mohamed G. Gouda. Distributed Reset. IEEE Trans. Computers 43(9): 1026-1038 (1994).

[4] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985.

[5] B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. Proc. of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing (PODC'94), pages 254-263, 1994.

[6] Baruch Awerbuch, Boaz Patt-Shamir, David Peleg, and Mike Saks. Adapting to asynchronous dynamic networks. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia*, pages 557–570, May 1992.

[7] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico*, pages 268–277, October 1991.

[8] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Bounding the Unbounded. Proc. 13th IEEE INFOCOM, June 1994, pp. 776-783.

[9] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-Stabilization by Local Checking and Global Reset (Extended Abstract). Proc. WDAG 1994, pp. 326-339.

[10] Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *31st Annual Symposium on Foundations of Computer Science*, 1990.

[11] Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. Maintaining Digital Clocks In Step. Parallel Processing Letters, Vol 1, 1991, pp. 11-18.

[12] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *29th Annual Symposium on Foundations of Computer Science*, pages 206–220, October 1988.

[13] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico*, pages 258–267, October 1991.

[14] J.E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.

[15] J.E. Burns, M.G. Gouda, and R.e. Miller. On relaxing interleaving assumptions. Proceedings of *the MCC workshop on Self Stabilizing Systems*, MCC Technical Report No. STP-379-89, 1989.

[16] Christian Boulinier, Franck Petit, Vincent Villain. When graph theory helps self-stabilization. Proc. ACM PODC 2004, St. John's, Newfoundland, Canada, pp. 150 - 159.

[17] S. Chandrasekar and PK Srimani. A self-stabilizing algorithm to synchronize digital clocks in a distributed system. Computers and Electrical Engineering, 20(6):439-444, 1994.

[18] A. Ciuffoletti. Self-stabilizing clock synchronization in a hierarchical network. In Proceedings of the Third Workshop on Self-Stabilizing Systems, pages 86-93, 1999.

[19] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. ACM*, 17:643–644, 1974.

[20] Shlomi Dolev. Possible and Impossible Self-Stabilizing Digital Clock Synchronization in General Graphs. Real Time Systems, 12, 95-107 (1997).

[21] Shlomi Dolev and Ted Herman. Superstabilizing Protocols for Dynamic Distributed Systems. Chicago J. Theor. Comput. Sci. 1997.

[22] Shlomi Dolev and Ted Herman. Parallel composition of stabilizing algorithms. Proc. WSS 1999, 25-32.

[23] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform Dynamic Self-Stabilizing Leader Election. IEEE Trans. Parallel Distrib. Syst. 8(4): 424-440 (1997).

[24] S. Dolev and J.L. Welch. Wait- Free clock synchronization. Algorithmica 18(4): 486-511 (1997).

[25] S. Dolev and J.L. Welch. Self- stabilizing clock synchronization in the presence of Byzantine faults. J. ACM 51(5): 780-799 (2004).

[26] S. Even and S. Rajsbaum. Shimon Even, Sergio Rajsbaum: Unison, Canon, and Sluggish Clocks in Networks Controlled by a Synchronizer. Mathematical Systems Theory 28(5): 421-435 (1995).

[27] S. Finn. Resynch Procedures and a Fail-Safe Network Protocol. IEEE Trans. on Commun., COM-27(6):840–845, June 1979.

[28] J-M. Couvreur, N. Francez, and M. Gouda. Asynchronous Unison. Proc. ICDCS, Yokohama, Japan, 1992, pp. 468-493.

[29] T. Herman. A stabilizing repair timer. Proc. DISC'98 Distributed Computing 12th International Symposium, Springer-Verlag LNCS:1499, pages 186-200, 1998.

[30] Ted Herman and Sukumar Ghosh. Stabilizing phase-clocks. Information Processing Letters, Volume 54, Issue 5 (June 1995), Pages: 259 - 265, 1995.

[31] M.G. Gouda and T. Herman. Stabilizing Unison. Information Processing Letters 35 (1990) pp. 171-175.

[32] Rachid Guerraoui and Marko Vukolic. How fast can a very robust read be? ACM PODC 2006.

[33] ST Huang and TJ Liu. Four-state stabilizing phase clock for unidirectional rings of odd size. Information Processing Letters, 65(6):325-329, 1998.

[34] ST Huang and TJ Liu. Self-stabilizing 2(m)-clock for unidirectional rings of odd size. Distributed Computing, 12:41-46, 1999.

[35] Shing-Tsaan Huang, Tzong-Jye Liu, Su-Shen Hung. Asynchronous Phase Synchronization in Uniform Unidirectional Rings. IEEE Trans. Parallel Distrib. Syst. 15(4): 378-384 (2004).

[36] S. S. Kulkarni and A. Arora. Multitolerant barrier synchronization. Information Processing Letters, vol 64(1), pp 29-36, October 1997.

[37] S. S. Kulkarni and A. Arora. Low-cost fault-tolerance in barrier synchronizations. International Conference on Parallel Processing, August 1998.

[38] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

Lin

[39] Chengdian Lin and Janos Simon. Possibility and impossibility results for self-stabilizing phase clocks on synchronous rings. Proceedings of the Second Workshop on Self-Stabilizing Systems, pp. 10.1–10.15,1995.

[40] Jeniffer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and Computation*, 62(2-3):190–204, 1984.

[41] Nancy Lynch. Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[42] D.L. Mills. Network Time Protocol: Specification and Implementation (version 1). University of Delaware, July 1988; RFC-1059, DARPA Network Working Group.

[43] J. Misra. Phase Synchronization. Information Processing Letters 38 (1991), pp. 101-105.

[44] Sen Moriya, Michiko Inoue, Toshimitsu Masuzawa, Hideo Fujiwara. SelfStabilizing WaitFree Clock Synchronization with Bounded Space. Proc. OPODIS 1998: 129-144.

[45] Amit Antil Nanavati. A simple self-stabilizing reset protocol. Symposium on Applied Computing archive Proceedings of the 1996 ACM symposium on Applied Computing, Philadelphia, Pennsylvania, United States (ISBN:0-89791-820-7). 1996, Pages: 93 - 97

[46] M. Papatriantafilou and P. Tsigas. On self-stabilizing wait-free clock synchronization. Parallel Processing Letters, 7(3):321-328, 1997.

[47] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. Comput.*, 18(2):740–747, 1989.

[48] John M. Spinelli and Robert G. Gallager. Broadcasting topology information in computer networks. *IEEE Trans. Comm.*, May 1989.

[49] Gerard Tel. Introduction to Distributed Algorithms (2nd ed.). Cambridge University Press, 2000.

[50] George Varghese. Self-Stabilization by Local Checking and Correction. PhD thesis, MIT Lab. for Computer Science, 1992.

[51] George Varghese. Self-Stabilization by Counter Flushing. SIAM J. Comput. 30(2): 486-510 (2000).

Biographies

Baruch Awerbuch is a professor of Computer Science at the Johns Hopkins University.

He received his Ph.D. in 1984 from the Technion, Haifa. From 1984 till 1994 he was at MIT as post-doc, research associate, and faculty at the Mathematics department.

His research interests are in the areas of algorithms, wireless networks, learning, security, and peer-to-peer systems with the emphasis on algorithmic approaches to real systems and networks

Shay Kutten received his PhD in Computer Science from the Technion, Israel, in 1987. From 1987 to 2000 he was with IBM T.J. Watson Research Center, as a post doctoral fellow, as a project leader, as the manager of the Network Architecture and Algorithms group and of Distributed Systems Security, and as a Research Staff Member. He led the network security project which developed the security architecture for several IBM products, and developed algorithms for network control, security, and distributed processing control, that were later used in IBM's products. He received (in 1993) the IBM Corporation Outstanding Innovation Award (OIA) for his work on distributed control protocols that were basis to the distributed control of IBM Broad Band Networking Services architecture, NBBS. In 1994 he received the IBM Outstanding Innovation Award (OIA) for his work on authentication protocols and his contributions to IBM's network security and NetSP (by itself an award winning product). The same authentication protocols influenced the later development of the Internet payment protocol, so IBM had to grant a no-fee license, so that the Internet can adopt this payment protocol. At the Technion he was the head of the Information Systems Area of the Davidson Faculty of IE&M at the Technion and the coordinator of undergraduate studies of the faculty of Industrial Engineering and Management. He won the Taub Award for excellence in research, and the Mitchner Award for research on Quality Sciences and Quality Management. He is an area editor (for security, reliability, and availability) of the ACM's journal on Selected Topics in Mobile Networks and Applications (MONET). He was also a member of the editorial board of the ACM's Wireless Networks and of the Elsevier journal Computer Networks and chaired program committees of the 1998 DISC conference and of the ACM PODC'04 conference.

Prof. Yishay Mansour Completed his BA and his MSc at the Technion in 1985 and 1987 and his PhD at MIT in 1990. Following that he was a postdoctoral fellow at Harvard University and a Research Staff Member at IBM T.J. Watson Research Center. Since 1992 he is with the School of Computer Science at Tel-Aviv University, where he was the chairman during 2000-2002.

Prof. Mansour has written numerous papers in all areas of computer science with special emphasis on communication networks, machine learning, reinforcement learning, computational game theory and algorithm design. He has supervised over dozen graduate students, served on numerous program committees and is currently an editor in four leading journals.

Boaz Patt-Shamir received his BSc from Tel Aviv University, his MSc from Weizmann Institute of Science, and in 1995 his PhD from MIT. He was an Assistant Professor at Northeastern University, and then joined the Department of Electrical Engineering in Tel Aviv University, where he directs the Distributed Computation Laboratory.

George Varghese worked at DEC for several years designing DECNET protocols and products (bridge architecture, Gigaswitch) before obtaining his Ph.D in 1992 from MIT. He worked from 1993-1999 at Washington University. He joined UCSD in 1999, where he currently is a professor of computer science. He won the ONR Young Investigator Award in 1996, and was elected to be a Fellow of the Association for Computing Machinery (ACM) in 2002.

Together with colleagues, he has 14 patents awarded in the general field of Network Algorithmics. Several of the algorithms he has helped develop have found their way into commercial systems including Linux (timing wheels), the Cisco GSR (DRR), and Microsoft Windows (IP lookups). He also helped design the lookup engine for Procket's 40 Gbps forwarding engine. He has written a book on building fast router and endnode implementations called "Network Algorithmics", which was published in December 2004 by Morgan-Kaufman. In May 2004, he co-founded NetSift Inc., where he was the President and CTO. NetSift is now part of Cisco Systems. From Aug 2005 to Aug 2006, he worked at Cisco Systems to help equip future routers and switches to detect traffic patterns to facilitate traffic measurement and real-time intrusion detection.