Self-Stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks

Ariel Daliot[1], Danny Dolev[1]*, and Hanna Parnas[2]

[1] School of Engineering and Computer Science, The Hebrew University of Jerusalem, Israel. {adaliot,dolev}@cs.huji.ac.il

[2] Department of Neurobiology and the Otto Loewi Minerva Center for Cellular and Molecular Neurobiology, Institute of Life Science, The Hebrew University of Jerusalem, Israel. hanna@vms.huji.ac.il

Abstract. We define the "Pulse Synchronization" problem that requires nodes to achieve tight synchronization of regular pulse events, in the settings of distributed computing systems. Pulse-coupled synchronization is a phenomenon displayed by a large variety of biological systems, typically overcoming a high level of noise. Inspired by such biological models, a robust and self-stabilizing pulse synchronization algorithm for distributed computer systems is presented. The algorithm attains near optimal synchronization tightness while tolerating up to a third of the nodes exhibiting Byzantine behavior concurrently. We propose that pulse synchronization algorithms can be suitable for a variety of distributed tasks that require tight synchronization but which can tolerate a bound variation in the regularity of the synchronized pulse invocations.

1 Introduction

The phenomenon of synchronization is displayed by many biological systems [27]. It presumably plays an important role in these systems. For example, the heart of the lobster is regularly activated by the synchronized firing of four interneurons in the cardiac ganglion [13, 14]. It was concluded that the organism cannot survive if all four interneurons fire out of synchrony for prolonged times [26]. This system inspired the present work. Other examples of biological synchronization include the *malaccae* fireflies in Southeast Asia where thousands of male fireflies congregate in mangrove trees; flashing in synchrony [4]; oscillations of the neurons in the circadian pacemaker; determining the day-night rhythm; crickets that chirp in unison; coordinated mass spawning in corals and even audience clapping together after a "good" performance [24]. Synchronization in these systems is typically attained despite the inherent variations among the participating elements, or the presence of noise from external sources or from participating elements. A generic mathematical model for synchronous firing of biological oscillators based on a model of the human cardiac pacemaker is given in [22]. This model does not account for either noise or the inherent differences among elements.

In computer science, synchronization is both a goal by itself and a building block for algorithms that solve other problems. In the "Clock Synchronization"

* This research was supported in part by Intel COMM Grant - Internet Network/Transport Layer & QoS Environment (IXA)

problem, it is desired for computers to have their clocks set as close as possible to each other as well as to keep a notion of real time (see [7, 25, 12, 18, 23]).

It is desired for algorithms to guarantee correct behavior of the network in face of faults or failing elements, sometimes irrespective of any initial state of the system (self-stabilization). It has been suggested in [26] that similar fault considerations may have been involved in the evolution of distributed biological systems. In the example of the cardiac ganglion of the lobster, it was concluded that at least four neurons are needed in order to overcome the presence of one faulty neuron, though supposedly one neuron suffices to activate the heart. The cardiac ganglion must be able to adjust the pace of the synchronized firing according to the required heartbeat, up to a certain bound, without loosing the synchrony (e.g. while escaping a predator a higher heartbeat is required – though not too high). Due to the vitality of this network, it is presumably optimized for fault tolerance, self-stabilization, tight synchronization and for fast re-synchronization.

The apparent resemblance of the synchronization and fault tolerance requirements of biological networks and distributed computer networks makes it appealing to infer from models of biological systems onto the design of distributed algorithms in computer science. Especially when assuming that distributed biological networks have evolved over time to particularly tolerate inherent in-homogeneity of the cells, noise and cell death. In the current paper, we show that in spite of obvious differences, a biological fault tolerant synchronization model ([26]) can inspire a novel solution to an apparently similar problem in computer science.

We propose a relaxed version of the Clock Synchronization problem, which we call "Pulse Synchronization", in which all the elements are required to invoke some regular pulse (or perform a "task") in tight synchrony, but allow for some variation from complete regularity. Though nodes need to invoke the pulses synchronously, there is a limit on how frequent it is allowed to be invoked (similar to the linear envelope clock synchronization limitation). The "Pulse Synchronization" problem resembles physical/biological pulse-coupled synchronization models [22], though in a computer system setting it is required to give an algorithm for the nodes to reach the synchronization requirement.

We present a novel algorithm in the settings of self-stabilizing distributed algorithms, instructing the nodes how and when to fire in order to meet the synchronization requirements of "Pulse Synchronization". The core elements of the algorithm are analogous to the neurobiological principles of *endogenous* (self generated) *periodic spiking, summation* and *time dependent refractoriness*. The basic algorithm is quite simple: every node invokes a pulse regularly and sends a message upon invoking it (*endogenous periodic spiking*). The node sums messages received in some "window of time" (*summation*) and compares this to the continuously decreasing time dependent firing threshold for invoking the pulse (*time dependent refractory function*). The node fires when the counter of the summed messages crosses the current threshold level, and then resets its cycle.

The refractory function is the key element of our algorithm for achieving fault tolerant synchronization.

The algorithm performs correctly as long as less than a third of the nodes behave in a completely arbitrary ("Byzantine") manner concurrently. It ensures a tight synchronization of the pulses of all correct nodes, while not using any cen-

tral clock or global pulse. We assume the physical network allows for a broadcast environment and has a bounded delay on message transmission. The algorithm may not reach its goal as long as these limitations are violated or the network graph is severely disconnected. The algorithm is Byzantine self-stabilizing and thus copes with a more severe fault model than the traditional Byzantine fault models. Traditional distributed algorithms assume the system is always within the predefined assumption boundaries. Moreover, many algorithms, particularly non-stabilizing clock synchronization algorithms, also make strong assumptions on the initial state of the nodes (such as assuming all clocks are initially synchronized, c.f. [7]). A self-stabilizing protocol converges to its goal within a finite time once the systems is back in an arbitrary state within the predefined assumption boundaries. For our protocol, once this happens, regardless of the state of the system, tight synchronization is achieved within finite time. It overcomes transient, permanent and intermittent faults, though for convergence we assume that there is a window of time during which no recovery of nodes takes place.

Our algorithm is uniform, all nodes execute an identical algorithm. It does not suffer from communication deadlock, as can happen in message-driven algorithms ([3]), since the nodes have a time-dependent state change, at the end of which they fire endogenously. It is fault-containing in the sense that the convergence time depends on the number of faulty nodes, f, and not on the network size (In Sect. 5 we discuss a relationship between the cycle length and the number of faults, imposed by our algorithm). The faulty nodes cannot ruin the synchronization; in the worst case, they can slow down the convergence and speed up the synchronized firing frequency up to a certain bound. The time complexity achieved is $O(f)$ cycles with a near optimal synchronization tightness of d (end to end network and processing delay). Comparatively, the time complexity of the digital clock synchronization problem in a similar model presented in [10] is exponential in the network size. The synchronization tightness achieved there depends on the network size. To the best of our knowledge that is the only paper describing an algorithm for Byzantine self-stabilizing clock synchronization algorithm. Note also that there are many papers that deal with self-stabilizing (digital) clock synchronization (see [17, 11, 1]), though not facing Byzantine faults. Nonetheless, the convergence time in those papers is not linear.

2 Assumptions, Definitions and Specifications of the Algorithm

The environment is a network of processors (nodes) that regularly invoke pulses, ideally every *cycle* time units. The invocation of the pulse is expressed by sending a message via a broadcast media; this is also referred to as **firing**.

In our environment, individual nodes have no access to a central clock or global pulse system. The hardware clocks of **correct nodes** have a bounded drift rate, ρ, from real time. The communication network does not guarantee any order on messages, though there is a bound on message delivery time once the network stabilizes.

The network and/or the nodes can behave arbitrarily, though eventually the network stabilizes to within the defined assumption boundaries in which at most f nodes may behave arbitrarily. The **network assumption boundaries** are:

1. Multicast message passing allowing for an authenticated identity of senders.
2. At most f of the nodes are faulty.
3. The subnetwork of correct nodes satisfies:
   (a) There is an upper bound δ on the physical end-to-end network delay.
   (b) Any message sent or received by any correct node will eventually reach every correct node within δ time units.

A node is **correct** at times that it complies with the following conditions:

1. Obeys a global constant $0 < \rho_{glob} << 1$, such that for every Newtonian time interval $[u, v]$, $(1 - \rho_{glob})(v - u) \leq$ 'physical clock' $\leq (1 + \rho_{glob})(v - u)$. Hereafter ρ_{glob} is denoted by ρ ($\rho \approx 10^{-6}$ in modern computers).
2. Operates according to an instructed protocol.
3. Accepts and sends only multicast messages.

Thus, a node is considered **faulty** when it violates one or more of the above. A faulty node can recover from its faulty behavior. The recovery is not necessarily immediate, but rather can take a certain amount of time until which the node is still considered faulty. We count the recovering nodes as part of the faulty nodes during a time window of length *cycle*. Thus, within the network assumption boundaries there can be at most f faulty and recovering nodes in every *cycle* time window. To overcome intermittent faults we need to assume some bound on how frequently the adversary can fail a recovered node, since, over time, all nodes may repeatedly fail and recover. We assume that eventually there is a window of at least $2(2f + 1)$ cycles within which there is no new recovery of nodes, which is proved in Theorem 3 to allow the system to converge.

Basic notations:

We use the following notations to define the quality of the solution, though nodes do not need to maintain them as variables.

- σ represents the target upper bound on the real time between the invocations of the pulses of different correct nodes (*tightness of synchronization*).
- $\Psi_p(t)$ is the number of endogenous (self generated) pulses node p would have invoked since its last recovery time to time t, had it been alone in the system.
- $\Phi_p(t)$ is the effective (actual) number of pulses a correct node p invoked since its last recovery time to time t.
- Let $a, b, g, h \in R^+$ be constants that define the linear envelope bound on the ratio between $\Psi_p(t)$ and $\Phi_p(t)$.
- $\phi_i \in R^+ \cup \{\infty\}$, $0 \leq i \leq n$, denotes the elapsed real time since the last time node p_i invoked a pulse. For a node, p_j, that has not fired since initialization of the system, $\phi_j \equiv \infty$.
- $d \equiv \delta + \pi$, where π is the upper bound on the message processing time.

Thus, d is an upper bound on the elapsed real time from the arrival of a message at a correct node and until a consequent message that is sent arrives at all other correct receiving nodes.

Basic definitions:

- The **state** of the system at time t is given by: $State \equiv (t, \phi_0, \ldots, \phi_{n-1})$.
- Let G be the set of all possible system states of a system S.

- A set of nodes, N, are called **synchronized** at time t if $\forall p_i, p_j \in N$, $\phi_i, \phi_j \leq \frac{cycle}{1-\rho}$, then
  1. $|\phi_i - \phi_j| \leq \sigma$, or
  2. $\frac{cycle}{1-\rho} - \sigma \leq |\phi_i - \phi_j| \leq \frac{cycle}{1-\rho}$.

  Note that every correct node is always synchronized with itself. If $\forall p_i, p_j \in N$, it is Condition 1 that holds then we say that N is **strongly synchronized.**
- $s \in G$ is a **synchronized state** of the system at time t if the set of correct nodes are strongly synchronized at some time t_{syn} in the interval $[t, t + \sigma]$.
- $s \in G$ is a **sub-synchronized state** of the set of nodes N at time t, if a set of correct nodes in N are strongly synchronized at some time t_{syn} in the interval $[t, t + \sigma]$.

Note that when $||N|| = n$ for a set of nodes N, then the two definitions coincide. Observe that the definition of synchronized nodes is not transitive, and therefore the definition of a (sub-)synchronized state is not transitive.

The Self-Stabilizing "Pulse Synchronization" Problem

As long as the system is within its assumption boundaries:

Convergence: Starting from an arbitrary state, s, the system reaches a synchronized state after a finite number of steps.
Closure: If s is a synchronized state of the system at time t_0 then

1. all subsequent system states are synchronized states,
2. $\ll$Linear Envelope$\gg$: for every correct node, p, and for every time $t \geq t_0$, $a[\Psi_p(t) - \Psi_p(t_0)] + b \leq \Phi_p(t) - \Phi_p(t_0) \leq g[\Psi_p(t) - \Psi_p(t_0)] + h$.

Note that the linear envelope resembles the linear envelope requirement of the clock synchronization problem.

3 The "Pulse Synchronization" Algorithm

We now present an algorithm that solves the "Pulse Synchronization" problem, inspired by and following a neurobiological analog. Each node sums the pulses that it learns about during a recent time window. If this sum (the Counter) crosses the current (time-dependent) threshold, then the node will fire. The refractory function describes the time dependency of the threshold. If reaching threshold level 0 then the node fires **endogenously,** irrespective of any inputs.

The refractory function
The cycle is the predefined time a correct node will count between two endogenous pulses. The refractory function R, is a step function. R_i is the time length of threshold level i. $R_i \in R^+$, $i = 1 \ldots n + 1$, where, $\sum_{i=1}^{n+1} R_i \equiv cycle$. R_{n+1} is called the **absolute refractory period** of the cycle. Using the neurobiological analogy, this is the first period after a node fires; the node never fires within an absolute refractory period. We assume that $cycle \gg \sigma$ and require that $R_{n+1} > 2\sigma$ (satisfied by Restriction 5 presented later). We denote $R^p(t)$ the threshold level of node p at time t or $R(t)$ if it is clear what node is referred to. If a node fires at time t, then R is **reset** to $R(t) = n + 1$.

The messages
The content of a message M_p from a node p, is a Counter - the number of recent inputs received that caused the sending node to fire. **A k-sized input** is a set of k such messages that are received within a certain time window (defined in the appendix) that causes the value of the Counter of the receiver to be at least k.

The Summation algorithm
A full description of the procedures used by the Summation Algorithm is provided in the Appendix. The Summation Algorithm is comprised of the following components:

The TIMELINESS procedure determines if the Counter contained in the message seems "reasonable" (timely). The bound on message transmission and processing time among correct nodes allows a node to estimate whether the content of a message it receives is plausible and therefore timely. If it is not then the node tables it for possible future use and after a certain time decays it. The MAKE-ACCOUNTABLE procedure determines by how much to increment the Counter following the arrival of a timely message. The node can take into account tabled messages, if necessary, for consistency reasons. The PRUNE procedure decays old messages and tables (currently) irrelevant messages.

SUMMATION(*at time* t_{event} *a new message* M_p *arrives* or *change in* R*)*
  if (*at time* t_{event} *a new message* M_p *arrives*) then
    if (TIMELINESS(M_p, t_{event})) then
      MAKE-ACCOUNTABLE(M_p)
      PRUNE(t);
  if (*change in* R) then
    PRUNE(t);

The event driven self-stabilizing algorithm (See Figure 1)

PULSE-SYNCHRONIZATION($n, f, cycle$)
INIT:
  if (n, f and $cycle$ are valid) then
    $R :=$ as determined by Eq. 2;
    reset R;
    set Counter $= 0$;
  else *exit;*
WAIT-FOREVER-LOOP:
  if (*at time* t_{event} *a new message* M_p *arrives* or *change in* R) then
    SUMMATION((M_p, t_{event}) or (*change in* R));
    if ($Counter \geq R(t)$) then
      Multicast Counter; / Invocation of the Pulse */*
      reset R;

4 Solving the "Pulse Synchronization" Problem

Due to the space constraints we only give an overview of the heuristics, in terms of the dynamics of the n-dimensional state space, that prove that the presented

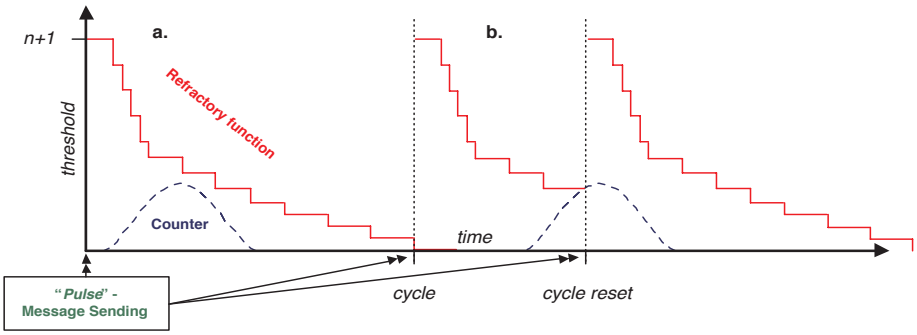

Fig. 1. Schematic example of the algorithm's mode of operation. (a.) A node fires endogenously if threshold=0 is reached. The counter of the summed messages does not cross the refractory function during the cycle, letting the threshold reach zero and consequently the node fires endogenously. (b.) The Counter of the summed messages increases sufficiently to cross the current refractory function, consequently the node fires and a new cycle of the node is initiated.

"Pulse Synchronization" algorithm solves the "Pulse Synchronization" problem. We assume the worst case of $n = 3f + 1$. In Sect. 5 we discuss various ratios between them and also about the relationship between the cycle length and f. A full account of the proofs of all the theorems in this paper can be found in [5].

1. Show that synchronized nodes identify a stable fixed point in state space (Lemma 1), which has a basin of attraction.
2. Show that we can associate every correct node exclusively with such a fixed point (called a "**synchronized cluster**") and prove that it is stable (Lemma 2), thus satisfying the closure requirement.
3. Show that the number of nodes associated with a synchronized cluster determines the span of its basin of attraction (depending on the shape of R). Explicitly formulate the basin of attraction of a synchronized cluster ("**absorbance distance**") as a function of its cardinality and R (see Theorem 1).
4. Show that every specific association of nodes with disjoint synchronized clusters has a matching linear constraint on R (Eq. 1). Satisfying this constraint under certain restrictions (Restrictions 1-5), ensures that there will be at least two synchronized clusters whose basins of attraction overlap in phase space ("within absorbance distance of each other") and thus synchronize (see Theorem 2).
5. Satisfying the set of all possible constraints (all possible associations of nodes with disjoint synchronized clusters) yields a refractory function that ensures Convergence of the system within finite time from an arbitrary state (see Corollary 2 and Theorem 3).
6. Present a solution to this set of linear equations on R (see Theorem 4).
7. Deduce from Corollary 2 and Theorem 4 that the algorithm thus solves the "Pulse Synchronization" problem (see Theorem 5).

We now cite the main lemmas, theorems, definitions and notations used:

- C_i – synchronized cluster number i.

• n_i − cardinality of C_i (i.e. number of correct nodes associated with synchronized cluster C_i).
• c − current number of synchronized clusters in the current state; $c \geq 1$.
• $dist(p,q) \equiv |\phi_p - \phi_q|$ is the **phase difference** between nodes a and b.
• $dist(C_i, C_j) \equiv dist$(first node in cluster C_i, first node in cluster C_j).

Restriction 1: The Summation algorithm counts input messages such that: Following the arrival of a message from a correct node, every correct receiving node's Counter is incremented to hold a value greater than the Counter in the message.
Restriction 2: R_i is a monotonic decreasing function $R_i \geq R_{i+1}, i = 1 \ldots n-1$.
Restriction 3: $R_i > 3d$, $i = 1 \ldots n - f - 1$.
Restriction 4: $R_i > \sigma(1+\rho) + \frac{\rho}{1+\rho} \sum_{j=1}^{n+1} R_j, i = 1 \ldots n$.
Restriction 5: $R_{n+1} \geq 2d(1+\rho)\frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1}$.

Lemma 1. *A set of correct nodes that are in a sub-synchronized state at some time t', remain in a sub-synchronized state as long as the network remains within its assumption boundaries.*

Lemma 2. *The set of nodes comprising a synchronized cluster at some time t remain in a sub-synchronized state as long as the network remains within its assumption boundaries.*

Corollary 1. *A synchronized cluster encompassing all the correct nodes implies the system is in a synchronized state, satisfying the objective of the Pulse synchronization problem.*

Theorem 1. *Given two synchronized clusters, C_i preceding C_j, Restriction 1 through Restriction 4 imply that if C_i fires due to a k-sized input ($0 \leq k \leq f$)[3] and* **prior** *to the firing of C_i,*

$$dist(C_i, C_j) \leq \frac{1}{1-\rho} \sum_{g=f+1}^{f+n_i} R_g - \frac{2\rho}{1-\rho^2} \sum_{g=f+1}^{n+1} R_g$$

then C_i absorbs C_j.

Theorem 2. *For n, f, cycle and a given clustering of $n-f$ correct nodes into $c > 1$ clusters at time t_0, for which Eq. 1 is satisfied, there will be at least one cluster that will absorb some other cluster by time $t_0 + (2\,cycle)$.*

[3] It is sufficient to limit the discussion to the case in which C_i receives a $k \leq f$-sized (faulty) input even though in the "real life" scenario C_i can by all accounts receive larger inputs. The justification for this assumption is that a greater than f sized input can actually be looked upon as though the f faulty nodes fired along with some cluster C_i, yielding an $f + n_i$-sized input. We therefore assume the discussed clusters are the clusters firing with or immediately following the k-sized (faulty) input, where $k \leq f$. The same reasoning is behind the consistency of the algorithm in case all/some of the faulty nodes actually behave correctly.

Theorem 3. *Within at most* $2(2f+1)$ *cycle time units the pulse synchronization algorithm reaches a synchronized state of the system.*

Corollary 2. *For a given cycle,* f *and* n*, finding a solution to the problem of "Pulse Synchronization" employing the proposed algorithm, reduces to solving the set of linear constraints on* R *determined by Eq. 1.*

$$\sum_{j=1}^{c-1}\sum_{i=f+1}^{f+n_j} R_i + \sum_{i=1}^{n_c} R_i \geq (1+\frac{2\rho}{1+\rho}(n-f))\, cycle, \text{where } \sum_{j=1}^{c} n_j = n-f \ . \quad (1)$$

Theorem 4. *Given*[4] $n > 3f+1$ *and cycle, the refractory function:*

$$R_i = \begin{cases} \frac{(1+\frac{2\rho}{1+\rho}(n-f))\, cycle}{n-f} & i = 1 \ldots n-f-1 \\ \frac{R_1 - R_{n+1} - \frac{2\rho}{1+\rho}(n-f)\, cycle}{f+1} & i = n-f \ldots n \\ 2d(1+\rho)\frac{(\frac{1+\rho}{1-\rho})^{n+3}-1}{(\frac{1+\rho}{1-\rho})-1} & i = n+1, \end{cases} \quad (2)$$

constitutes a solution to the set of linear equations of R*.*

Theorem 5. *The algorithm of Sect. 3 solves the Pulse Synchronization Problem.*

5 Analysis of the Algorithm and Comparison to Related Algorithms

Our algorithm does not depend explicitly on the actual behavior of the nodes. The convergence of the clusters depends on having at most f nodes that may behave arbitrarily. Therefore, our algorithm can withstand any faulty behavior, including dynamic failure and recovery of nodes (intermittent faults). Nodes that have just recovered need time to synchronize, therefore, we assume that eventually we have a window of time within which there are no newly recovered nodes, and within which the system inevitably converges (Theorem 3).

Authentication and fault ratio: The algorithm does not require the full power of unforgeable signatures, and only the equivalence of an authenticated channel is required. Note that the shared memory model ([10]) has an implicit assumption that is equivalent to an authenticated channel, since a node "knows" the identity of the node that wrote to the memory it reads from. A similar assumption is also implicit in many message passing models by assuming a direct link among neighbors, and as a result, a node "knows" the identity of the sender of a message it receives.

Many fundamental problems in distributed networks have been proven to require $3f+1$ nodes to overcome f concurrent faults in order to reach a deterministic solution without authentication [15, 20, 7, 6]. We did not prove that this ratio is a necessary requirement for solving the "Pulse Synchronization" problem

[4] An equivalent proof to Theorem 4 is given for the case $n = 3f+1$ in [5].

but results of related problems lead to believe that a similar result may exist for this problem.

There are algorithms that have no lower bound on the number of nodes required to handle f faults, but unforgeable signatures are required as all the signatures in the message are validated by the receiver [7]. This is costly time-wise, it increases the message size, and it introduces other limitations, which our algorithm does not have.

Time complexity: A randomized self-stabilizing Byzantine digital clock synchronization algorithm in [9, 10] designed for a fully connected communication graph synchronizes in $M2^{2(n-f)}$ pulses, where M is the upper bound on the clock values held by individual processors. The algorithm uses broadcast message passing, it allows transient and permanent faults during convergence, requires at least $3f + 1$ processors, but utilizes a global pulse system. An additional algorithm in [10], does not use a global pulse system and is thus partially synchronous similar to our model. The convergence time of the latter algorithm is $O((n - f)n^{6(n-f)})$. This is drastically higher than our result even if one takes into consideration the relationship between *cycle* and f of our solution(Eq. 3), which yields an $O(f^3)$ time units convergence complexity.

Our use of broadcasts is essentially similar to the type of broadcast used in [10], since we assume nothing on the order of arrival of the messages and concomitant messages from the same source are ignored. Thus, a faulty node can send two consecutive messages with differing values. Correct nodes will accept the first arriving one.

Message and space complexity: Each correct node multicasts exactly one message per cycle. This yields a message complexity of at most n messages per cycle. The system's message complexity to reach synchronization from any arbitrary state is at most $2n(2f + 1)$ messages per synchronization from any arbitrary initial state. The faulty nodes cannot cause the correct nodes to fire more messages during a cycle. Comparatively, the self-stabilizing clock synchronization algorithm in [10] sends n messages during a pulse and thus has a message complexity of $O(n(n - f)n^{6(n-f)})$. This is significantly larger than our message complexity irrespective of the time interval between the pulses.

The space complexity is $O(n)$ since the variables maintained by the processors keep only a linear number of messages recently received and various other small range variables. The number of possible states of a node is linear in n, but the node does not need to keep a configuration table.

Tightness of synchronization: Using the presented algorithm, the invocation of the pulses of the nodes will be synchronized to within d real time units. This asymptotically equals the lower bound on clock synchronization $d(1 - 1/n)$ in completely connected, fault-free networks [19]. Were our algorithm to be used as a logical clock synchronization algorithm then the clocks would show a maximum time difference of $d(1 + \rho)$ between each other due to the hardware clock skew. Comparatively, the non-stabilizing clock synchronization algorithm of [7] reaches a synchronization tightness of $d(1 + \rho) + 2\rho(1 + \rho)R$ where R is the time between synchronization rounds. The Byzantine digital clock synchronization algorithm in [10], which does not assume a global pulse, reaches a synchronization tightness which is in the magnitude of $(n - f)d(1 + \rho)$. This is significantly less tight than our result.

Firing frequency bound ("Linear Envelope"): The firing frequency bound is around twice that of the original firing frequency of the nodes (linear envelope). This bound is determined by the fault ratio (the sum of the first f threshold steps relative to the cycle length). For $n = 3f + 1$ it translates to $\approx \frac{1}{2}cycle$. Thus, if required, the firing frequency bound can be closer to the endogenous firing frequency of 1 *cycle* if the fraction of faulty nodes is assumed to be lower. For example, for a fraction of $n = 10f$, the lower bound on the cycle length is approximately $8/9$ that of the endogenous cycle length.

The allowed parameter range (of n, f and $cycle$): Restrictions 2-5 determine the lower bound on the allowed length of each threshold step, R_i. There are exactly $n + 1$ threshold steps whose sum needs to equal exactly $cycle$ time units. Restrictions 2-5 therefore also determine a required relationship between the input parameters n, f and $cycle$. To uncover this relationship and show that the solution space is not empty we need to verify that given $n, f, cycle$ and the solution specified by Eq. 2 then:

1. R_i, $i = 1 \ldots n - f - 1$, comply with Restrictions 3 and 4:
$$\frac{(1 + \frac{2\rho}{1+\rho}(n - f))\, cycle}{n - f} > \max(3d, d(1 + \rho) + \frac{\rho}{1 + \rho}\, cycle) \ .$$
2. R_i, $i = n - f \ldots n$, comply with Restriction 4:
$$\frac{R_1 - R_{n+1} - \frac{2\rho}{1+\rho}(n - f)\, cycle}{f + 1} > d(1 + \rho) + \frac{\rho}{1 + \rho} \sum_{j=n-f}^{n+1} R_j \ .$$
3. R_i, $i = 1 \ldots n - f - 1$ comply with: $R_i > \sum_{j=n-f}^{n+1} R_j$.

This yields the following approximate required relationship between f and $cycle$ (assuming $n = 3f + 1$), ensuring that the solution of (Eq. 2) complies with Restrictions 2-5:

$$4df^2 < cycle \ . \tag{3}$$

A disadvantage of the algorithm is thus its scalability with respect to the fault ratio and cycle length should the fault ratio be taken at its bound, $n = 3f + 1$. This relationship improves though, with a decrease in d, the network delay. If we fix the solution for a boundary network size of $3f + 1$ and designate it as m, then executing the algorithm with any input network size of $n \geq m$ is acceptable.

6 Discussion

We developed a self-stabilizing fault tolerant algorithm for *pulse synchronization* in distributed computer networks. The "Pulse Synchronization" problem is a relaxed version of the clock synchronization problem. The linear envelope requirement defines a restriction that defies a straight forward simple solution. Moreover, the fact that there isn't a specific value that describes the pulse difference among nodes, eliminates the ability to use some sort of approximation approach ([8]) or consensus. Clock synchronization can be viewed as approximate agreement on a value representing real time.

The algorithm developed is inspired by and shares properties with the lobster cardiac pace-maker network; the network elements fire in tight synchrony

within each other, whereas the synchronized firing pace can vary, up to a certain extent, within a linear envelope of a completely regular firing pattern. This behavior can be compared to the requirement of keeping the clocks of the network computers synchronized with each other ("precision") as well as being close to real time ("accuracy"). Pulse synchronization and clock synchronization have similar precision requirements but differ in the accuracy requirement.

Possible applications: We offer the pulse synchronization scheme in general, and our Byzantine self-stabilizing pulse synchronization algorithm in particular. The algorithm has an improved fault tolerance (self-stabilization) in comparison to traditional clock synchronization algorithms and improved convergence time and synchronization tightness in comparison to Byzantine self-stabilizing digital clock synchronization. It is useful particularly when there is no real need to agree on the pulse counters or keeping a constant pulse frequency, but only obtain and maintain highly fault tolerant and tight synchronization.

A self-stabilizing pulse synchronization algorithm can be beneficial to solve TDMA based slot synchronization (in multi access channels such as cellular phones; slotted ALOHA) [28]. A similar suggestion appears in [21], though without supplying any fault tolerance. In this problem, if the participants are not tightly synchronized with respect to the frame starting point then collisions will occur. The exact starting time of the frame is of minor importance relative to the synchronization requirement. Should a central synchronizer be present then the network would be extremely sensitive to faults in the central station. Where no central station is available and connectivity changes rapidly a self-stabilizing distributed synchronization protocol would make the participants resynchronize without any exterior intervention, should the participant be out of synchronization. Other cases that could benefit from a self-stabilizing pulse synchronization scheme include global snapshots; load balancing; distributed inventory count; distributed scheduling; backup of distributed systems; debugging of distributed programs; and deadlock detection.

References

1. A. Arora, S. Dolev, and M. G. Gouda, "*Maintaining digital clocks in step*", Parallel Processing Letters, 1:11-18, 1991.
2. G. E. Andrews, "*The Theory of Partitions*", Encyclopedia of Mathematics and Its Applications, Vol. 2, Addison-Wesley, Reading, MA, 1976.
3. J. Brzeziński, and M. Szychowiak, "*Self-Stabilization in Distributed Systems - a Short Survey*, Foundations of Computing and Decision Sciences, 25(1), 2000.
4. J. Buck, and E. Buck, "*Synchronous fireflies*", Scientific American, Vol. 234, pp. 74-85, May 1976.
5. A. Daliot, D. Dolev, H. Parnas, "*Self-Stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks* Technical Report TR2003-1, Schools of Engineering and Computer Science, The Hebrew University of Jerusalem, March 2003. url://leibnitz.cs.huji.ac.il/tr/acc/2003/HUJI-CE-LTR-2003-1_pulse-tr6.ps
6. D. Dolev, J. Halpern, and H. R. Strong, "*On the Possibility and Impossibility of Achieving Clock Synchronization*", J. of Computer and Systems Science, Vol. 32:2, pp. 230-250, 1986.
7. D. Dolev, J. Y. Halpern, B. Simons, and R. Strong, "*Dynamic Fault-Tolerant Clock Synchronization*", J. Assoc. Computing Machinery, Vol. 42, No.1, pp. 143-185, Jan. 1995.

8. D. Dolev, N. A. Lynch, E. Stark, W. E. Weihl and S. Pinter, "*Reaching Approximate Agreement in the Presence of Faults*, Journal of the ACM, 33 (1986).
9. S. Dolev, "*Self-Stabilization*," The MIT Press, 2000.
10. S. Dolev, and J. L. Welch, "*Self-Stabilizing Clock Synchronization in the presence of Byzantine faults*", Proc. Of the Second Workshop on Self-Stabilizing Systems, pp. 9.1-9.12, 1995.
11. S Dolev and JL Welch, "*Wait-free clock synchronization*", Algorithmica, 18(4):486-511, 1997.
12. C. Fetzer and F. Cristian, "*An Optimal Internal Clock Synchronization Algorithm*", Proceedings of the 10^{th} Conference on Computer Assurance, 1995, pp. 187-196, Gaithersburg, MD, USA.
13. W. O. Friesen, "*Physiological anatomy and burst pattern in the cardiac ganglion of the spiny lobster Panulirus interuptus*", J. Comp. Physiol., Vol. 101, 1975.
14. W. O. Friesen, "*Synaptic interaction in the cardiac ganglion of the spiny lobster Panulirus interuptus*", J. Comp. Physiol., Vol. 101, pp. 191-205, 1975.
15. M. J. Fischer, N. A. Lynch and M. Merritt, "*Easy impossibility proofs for distributed consensus problems*", Distributed Computing, Vol. 1, pp. 26-39, 1986.
16. T. Herman, "*Phase clocks for transient fault repair*", IEEE Transactions on Parallel and Distributed Systems, 11(10):1048-1057, 2000.
17. T. Herman and S. Ghosh, "*Stabilizing Phase-Clocks*", Information Processing Letters, 5(6):585-598, 1994.
18. B. Liskov, "*Practical Use of Synchronized Clocks in Distributed Systems*", PODC 10, 1991, pp. 1-9.
19. J. Lundelius, and N. Lynch, "*An Upper and Lower Bound for Clock Synchronization*," Information and Control, Vol. 62, pp. 190-205, Aug/Sep. 1984.
20. N. Lynch, "*Distributed Algorithms*", Morgan Kaufmann, 1996.
21. R. Mathar and J. Mattfeldt, "*Pulse-coupled decentral synchronization*", SIAM J. Appl. Math, Vol. 56, No. 4, pp. 1094-1106, Aug. 1996.
22. R. E. Mirollo and S. H. Strogatz, "*Synchronization of pulse-coupled biological oscillators*", SIAM J. Appl. Math, Vol. 50, pp. 1645-1662, 1990.
23. B. Patt-Shamir, "*A Theory of Clock Synchronization*, Doctoral thesis, MIT, Oct. 1994.
24. Z. Nèda, E. Ravasz, Y. Brechet, T. Vicsek, and A.-L. Barabàsi, "*Self-organizing process: The sound of many hands clapping*", Nature, 403, pp. 849-850, 2000.
25. F. Schneider, "*Understanding Protocols for Byzantine Clock Synchronization*", Technical Report 87-859, Dept. of Computer Science, Cornell University, 1987.
26. E. Sivan, H. Parnas and D. Dolev, "*Fault tolerance in the cardiac ganglion of the lobster*", Biol. Cybern., Vol. 81, pp. 11-23, 1999.
27. S. H. Strogatz and I. Stewart, "*Coupled Oscillators and Biological Synchronization*", Scientific American, Vol. 269, pp. 102-109, Dec. 1993
28. A. S. Tanenbaum, "*Computer Networks* 3^{rd} *ed.*", Prentice Hall International.

A Appendix: The Summation Algorithm

Heuristics:

1. When the input counter crosses the threshold level, either due to a sufficient counter increment or a threshold decrement, then the node sends a message (fires). The message sent holds the value of Counter at sending time.
2. The TIMELINESS procedure is employed at the receiving node to assess the credibility (timeliness) of the value of the Counter contained in this message. This procedure ensures that messages sent by correct nodes with Counter less than n will always be assessed as timely by other correct receiving nodes.

3. When a received message is declared timely and therefore accounted for it is stored in a "counted" message buffer ("Counted State"). The receiving node's Counter is then updated to hold a value greater than the Counter in the message by the MAKE-ACCOUNTABLE procedure.
4. If a message received is declared untimely then it is temporarily stored in an "uncounted" message buffer ("Uncounted State") and will not be accounted for at this stage. Over time, the timeliness test of previously stored timely messages may not hold any more. In this case, such messages will be moved from the Counted State to the Uncounted State by the PRUNE procedure.
5. All messages are deleted after a certain time-period by the PRUNE procedure.

Definitions and state variables

Counter: an integer representing the node's estimation of the number of timely firing events received from distinct nodes within a certain time window. Counter is updated upon receiving a timely message. The node's Counter is checked against the refractory function whenever one of them changes. The value of Counter is bounded and non-monotonous; the arrival of timely events may increase it and the decay/untimeliness of old events may decrease it.
Signature entry: a basic data structure represented as (S_p, t_{arr}) and created upon arrival of a message M_p. S_p is the id (or signature) of the sending node p and t_{arr} is the arrival time of the message. We say that two signature entries, (S_p, t_1) and (S_q, t_2), are **distinct** if $p \neq q$.
Counted State (CS): a set of distinct signature entries that determine the current value of Counter. The Counter reflects the number of signature entries in the Counted State. A signature entry is **accounted for** in Counter, if it was in CS when the current value of Counter was determined. We require exclusive write access to CS.
Uncounted State (UCS): a set of signature entries, not necessarily distinct, that have not been accounted for in the current value of Counter and that are not yet due to decay. A signature entry is placed in the UCS when its message clearly reflects a faulty sending node or because it is not timely anymore.
Retired UCS (RUCS): a set of distinct signature entries not accounted for in the current value of Counter due to the elapsed time since their arrival. These signature entries are awaiting deletion (decaying).

The CS and UCS are mutually exclusive and together reflect the relevant messages received from other nodes in the preceding time window. Their union is the node's **message state.**

We use the notation $Counter_q$ to mark node q's Counter, $Counter_{M_p}$ to mark the Counter contained in a message M_p sent by p and $|\text{message state}|$ to mark the number of distinct signature entries in the message state.
• $T(p, M_p)$: denotes the time at node p at which it sent the message M_p.
• $T(M_p, q)$: denotes the arrival time of the message M_p at node q.
• $MessageAge(t, p)$: the elapsed time on a node's clock since the most recent arrival of a message from node p. Thus, its value at a node q at time t, is given by $t - T(M_p, q)$.
• $StateAge(t, k)$: denotes $MessageAge(t, \ldots)$ of the k'th most recent signature entry in the node's message state.
• $CSAge(t)$: denotes, at t, the largest $MessageAge(t, \ldots)$ among the signature entries in CS.

Procedures used by the Summation algorithm

We say that message M_p is **assessed** by q, once the following procedure is completed by q. A message M_p, is **timely** at time t_{arr} at node q once it is declared timely by the procedure.

TIMELINESS (M_p, t_{arr}):

Timeliness Condition 1:

If $(0 > Counter_{M_p}$ or $Counter_{M_p} > n-1)$
then return "M_p is not timely";

Timeliness Condition 2:

Create a new signature entry (S_p, t_{arr}) and insert it into UCS;
If $(\exists (S_p, t)$ s.t. $t \neq t_{arr})$[5] in message state $\cup RUCS)$
then
delete from message state all (S_p, t'), where $t' \neq \max(T(M_p,q))$;
return "M_p is not timely";

Timeliness Condition 3:

Let k denote $Counter_{M_p}$, and let $\tau(k) \equiv 2d(1+\rho)\frac{(\frac{1+\rho}{1-\rho})^{k+1}-1}{(\frac{1+\rho}{1-\rho})-1}$.
If $(k <$ |message state|) AND $(StateAge(t_{arr}, k+1) \leq \tau(k))$
at some time in the interval $[t_{arr}, t_{arr} + d(1+\rho)]$[6]
then return "M_p is timely";
else return "M_p is not timely";

The following procedure atomically moves and deletes obsolete signature entries. It prunes the CS to hold only signature entries such that a message sent holding the resultant Counter will be assessed as timely at any correct receiving node. Counter is then updated.

PRUNE(t)**:**

- Delete (decay) from RUCS all entries (S_p, t) whose $MessageAge(t, p) > \tau(n+2)$;
- Move to RUCS, from the message state, all signature entries (S_p, t) whose $MessageAge(t, p) > \tau(n+1)$;
- Move to UCS, from CS, signature entries, beginning with the oldest, until: $CSAge(t) \leq \tau(k-1)$, where $k = \max(1, |CS|)$;
- Set $Counter = |CS|$;

This procedure atomically moves signature entries from UCS into CS and updates the value of Counter following the arrival of a new timely message M_p.

MAKE-ACCOUNTABLE(M_p**):**

- Move the $\max(1, Counter_{M_p} - Counter_q + 1)$ most recent distinct signature entries from UCS to CS;
- Set $Counter = |CS|$;

This set of procedures comprise the event driven Summation Algorithm.

[5] We assume no concomitant messages are stamped with the exact same arrival times at a correct node. We assume that one can uniquely identify messages.

[6] We assume the implementation detects that these conditions are satisfied within the time window.